Linear Approximations For Factored Markov Decision Processes

by

Relu-Eugen Patrascu

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Doctor of Philosophy
in
Computer Science

Waterloo, Ontario, Canada, 2004

AUTHOR'S DECLARATION FOR ELECTRONIC SUBMISSION OF A THESIS

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

A Markov Decision Process (MDP) is a model employed to describe problems in which a decision must be made at each one of several stages, while receiving feedback from the environment. This type of model has been extensively studied in the operations research community and fundamental algorithms have been developed to solve associated problems. However, these algorithms are quite inefficient for very large problems, leading to a need for alternatives; since MDP problems are provably hard on compressed representations, one becomes content even with algorithms which may perform well at least on specific classes of problems. The class of problems we deal with in this thesis allows succinct representations for the MDP as a dynamic Bayes network, and for its solution as a weighted combination of basis functions. We develop novel algorithms for producing, improving, and calculating the error of approximate solutions for MDPs using a compressed representation.

Specifically, we develop an efficient branch-and-bound algorithm for computing the Bellman error of the compact approximate solution regardless of its provenance. We introduce an efficient direct linear programming algorithm which, using incremental constraints generation, achieves run times significantly smaller than existing approximate algorithms without much loss of accuracy. We also show a novel direct linear programming algorithm which, instead of employing constraints generation, transforms the exponentially many constraints into a compact form more amenable for tractable solutions. In spite of its perceived importance, the efficient optimization of the Bellman error towards an approximate MDP solution has eluded current algorithms; to this end we propose a novel branch-and-bound approximate policy iteration algorithm which makes direct use of our branch-and-bound method for computing the Bellman error. We further investigate another procedure for obtaining an approximate solution based on the dual of the direct, approximate linear programming formulation for solving MDPs. To address both the loss of accuracy resulting from the direct, approximate linear program solution and the question of where basis functions come from we also develop a principled system able not only to produce the initial set of basis functions, but also able to augment it with new basis functions automatically generated such that the approximation error decreases according to the user's requirements and time limitations.

Acknowledgements

First I would like to thank the people who had the most direct influence on this thesis: my advisor, Dale Schuurmans whose guidance, help, and relentless optimism has been invaluable; Pascal Poupart, for his insightful inputs and willingness to discuss any research, no matter how outlandish it may have appeared, and for being a good friend.

In addition, I would like to thank my committee: Ronald Parr, Robin Cohen, Brendan Frey, and Fakhri Karray for their excellent comments and suggestions. I consider myself a lucky man to have crossed paths with Craig Boutilier, Peter van Beek, Pascal Poupart, and Brendan Frey, who have been kind to me on more than one occasion. I also thank all my collaborators, including Carlos Guestrin, Jody Moran, and Fletcher Lu. I am indebted to my friends who contributed to making my graduate studies an unforgettable experience: Finnegan, Mihaela, Dana, Ali, Trausti, Chris, Fuchun, Yann, Howard, Amélie, Dave, and many more.

Finally, I would like to thank Valentina for her continuous support.

For my parents

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Classical algorithms from operations research such as value iteration, policy iteration, and linear programming [2, 30, 13, 51], have established the fundamental solutions for Markov Decision Processes (MDP), the model of choice in sequential decision making under uncertainty. These algorithms are ingeniously based on robust theory and one cannot easily find reasons to complain when solving problems with hundreds, thousands, and even hundreds of thousands of states. But at the core of these methods lies an explicit representation of the MDP, and therefore, when the problem size is significantly larger, with state spaces in the millions or more, they are quickly rendered with unacceptably long computation times. The three algorithms assume a table representation of the values involved; the table obligingly is large in size—at most polynomial in the state space size—as it needs to store values associated with all of the MDP states (e.g. immediate rewards and transition probabilities). In contrast with the explicit representation for which usually no assumption is made about the structure of the MDP—known or unknown— is the compact representation: it pre-supposes that internal structure exists which, if taken advantage of, hopefully results in significant computational savings.

Together with structure in the MDP and compact representation of the MDP components one would also prefer a compact representation of the required solution; but even then, when all these wishes are fulfilled, it is still an open question how to best take advantage of the seemingly helpful features a problem may exhibit. Since the desire to solve large systems has been present from the beginning, several proposals have been put forward; some whose underlying theme is divide-and-conquer—the original weakly coupled problem is decomposed into a few smaller problems which are then solved individually [14, 49, 36, 37]. Other notable approaches include exact and approximate solutions employing succinct piece-wise constant (decision trees) and linear representations [10, 56, 11, 25, 34, 35, 27, 28]. Our approach is similar in vein to the latter cited research, in that we seek to obtain efficient linear approximations in compressed representation MDPs.

It is not clear, however, that such solutions are applicable in all cases and whether some solutions are always and in general superior to others. What one would expect—and now seems to become clearer—is that algorithms which target specific structural assumptions will outperform more general algorithms in a practical setting. It should be noted that computational complexity results [46, 39, 43, 23, 22, 40, 41] deemed many of the interesting problems with compressed representation hard if either an exact solution or a good approximation is sought.

This thesis carries out an investigation into the use and further development of efficient approximate algorithms for factored MDPs which, not surprisingly, have already shown promising results [27, 28, 53, 47]. The family of algorithms we develop is based on the crafty application of *linear programming* [13] to the problem of approximating the value function of a given structured MDP. Linear programming has a long history and is well understood and used in operations research, economics, and other applied fields, but less so in the context of structured MDPs with succinct representation.[1]

We employ a linear combination of basis functions as the approximation architecture, and use various optimization tricks to solve the factored MDPs approximately. More specifically, the contributions of this thesis are:

- an efficient branch-and-bound algorithm for computing the Bellman error of the compact approximate solution obtained by solving a compactly represented MDP;
- by generating constraints as needed, an efficient direct linear programming algorithm with run time significantly smaller than existing approximate policy iteration algorithms;
- an efficient, direct linear programming algorithm using a transformation to re-write the exponentially many constraints into a compact form more amenable for tractable solutions;
- through a different use of our branch-and-bound search employed in calculating the Bellman error, an efficient approximate policy iteration algorithm which minimizes the Bellman error;
- an investigation into the use of the linear programming dual to obtaining an approximate solution;
- a principled system of producing the initial set of basis functions, or otherwise augmenting it with new basis functions such that the approximation error decreases.

The thesis is laid out as follows: Chapter 2 introduces notation and mathematical preliminaries describing the problem formally, and provides background on: compact representation of MDPs and their solutions, computational complexity results for exact and compactly

[1] We will use interchangeably the following expressions:*factored MDPs*, *MDPs with succinct representation*, and *compactly represented (structured) MDPs.*

represented MDPs (to whet the appetite), brief overview of methods for solving MDPs, both classical exact algorithms and newer approximate algorithms; this chapter provides material helpful in understanding the contributions and in setting a context for the following chapters. Chapter 3 presents some results concerning the hardness of calculating the Bellman error and the first contribution, an algorithm to compute this error efficiently for medium-size problems. The theme of Chapter 4 is the approximate solution of MDPs given a fixed set of basis functions; we describe our contributions on this subject. In Chapter 5 we continue by providing a novel suite of techniques that can be used to improve an approximate solution of a compactly represented MDP. We conclude with some short ending remarks in Chapter 6. Additionally, in an attempt to establish a smoother readability flow in the main text, we provide a description of the problems we used in simulations in Appendix A, and a complete suite of results from experiments related to Chapter 5 in Appendix B.

Chapter 2

General Preliminaries and Background

Particular features of an MDP problem make it belong to some class or another: on the one hand it is the process itself which can be of several different types; on the other hand it is the objective to be achieved that will typify the problem. Either way MDPs have well defined components and they are introduced in the next few sections.

Specifically, Sections 2.1 and 2.2 will introduce the MDP mathematical model, Sections 2.3 and 2.4 cover background material related to setting up an MDP problem and objective to optimize, while Section 2.5 deals with some terminology. Section 2.6 gives a brief overview of complexity results for problems with a flat representation. In Section 2.8 we introduce structured MDPs and their succinct representation, followed by an overview of complexity results specific to them in Section 2.9.

2.1 Markov Decision Processes

This section will introduce Markov decision processes and associated problems. It should serve mostly as a vehicle to establish notation, identify the types of MDPs of interest, and provide a quick overview of important facts from the theory of MDPs. Recommended in-depth treatments can be found in several books [2, 30, 21, 51, 4].

A Markov decision process is a mathematical model for describing systems in which long term behavior is to be optimized. Imagine the situation presented in Example 2.1.1.

Example 2.1.1

Mary decides that she should optimize her personal income by asking her parents for money more intelligently than she has done in the past. She knows her parents well, and has noticed the following.

- Father is more generous with money. When he is in a good mood, he gives Mary $2 if she asks him for money, gives Mary $1 if he overhears Mary asking mother for money, and gives her no money when he is not in a good mood.
- Mother is more careful with money, but also more merciful. When Mary asks her, she gives $1.5 when in a good mood and $0.5 when not in a good mood. However, when she overhears Mary asking Father for money, she gives her nothing.

- If Mary does not ask for money at all, her father gives her $0.5 in a good mood, and nothing otherwise.
- Mary's parents' future mood is also affected by her currently asking for money or not. Her father's mood is more likely to worsen by next day if Mary asked for money, than her mother's mood.

Mary also is of the opinion that money is more valuable in the present than in the future.

Given that she recently has taken an introductory course in operations research, she decides to apply her knowledge to solving this problem.

As we describe a Markov decision process mathematically, we will also use the simple example above as an illustration of the components involved.

A Markov decision process is defined as a tuple $< S, A, R, P, \nu >$. The set of states S can be either finite or infinite, but in this work only finite state spaces are considered, and we assume its size is $|S| = N$.

In our tiny example, the state space is comprised of four possible states, i.e. $S = \{x_1, x_2, x_3, x_4\}$. Their semantics are:

x_1 neither parents are in a good mood,

x_2 mother is in a good mood, but father is not,

x_3 father is in a good mood, but mother is not, and

x_4 both parents are in a good mood.

Similarly, the total number of actions is assumed finite $|A| = K$; actions are chosen and executed from a subset $A(s_t) \subseteq A$. In this manner we can define Mary's actions as $A = \{ask\_mother, ask\_father, ask\_neither\}$.

The model intends to describe how a process evolves over time, moving from one state to another after executing an action and receiving a reward. We only consider discrete time MDPs, hence transitions from one state to the next is done in stages.

We assume an initial stage, $t = 0$, from which the process starts in state $s_0 \in S$ with probability $\nu(s_0)$. This probability is commonly referred to as the *prior probability* over states; an explicit representation for ν is a $N \times 1$ vector. At all other stages $t = 1, 2, \ldots$ the process is in some state $s_t \in S$. One can also think of the sequence $s_0, s_1, \ldots$ as a

sequence of random variables. If the process has perfect knowledge of the actual current state, then the MDP is called *fully observable*; otherwise it is a *partially observable* MDP (POMDP).[1] The following—not necessarily ordered—events are associated with each stage $t = 1, 2, \ldots$:

- transition from the previous state s_{t-1} ends and the process arrives in the current state s_t,[2]
- the stage count is incremented from $t-1$ to t,
- the process receives a reward $r_t = R(s_{t-1}, a_{t-1}),$ where $R : S \times A \to [R_{min}, R_{max}]$,
- the process incurs a cost $c_t = C(s_{t-1}, a_{t-1}),$ where $C : S \times A \to [C_{min}, C_{max}]$ for executing action a_{t-1},[3]
- the process chooses and executes an action $a_t \in A$
- transition to the next state begins.

As mentioned in the introduction, state-to-state transitions are governed by transition probabilities

$$P(s_t|s_{t-1}, a_{t-1}, s_{t-2}, a_{t-2}, \ldots, s_1, a_1, s_0, a_0) = P(s_t|s_{t-1}, a_{t-1}). \tag{2.1}$$

Equation 2.1 makes the Markov property explicit: the transition probabilities are functions only of the present state and subsequent action. An explicit representation for these probabilities would be one $N \times N$ stochastic matrix (i.e. each row sums up to 1) P^a for each possible action a. R and C can be represented explicitly as an $N \times K$ matrix each.

Throughout the document quantities which have a physical aspect, such as vectors and matrices are denoted by bold types, (e.g. $\boldsymbol{\nu}$, $\boldsymbol{R}$, $\boldsymbol{P}$, etc.); when the the physical aspect is irrelevant, for instance when referring to sets, notation is done via plain types (e.g. ν, R, P, etc.); single elements are accessed in both cases using parentheses, as in $\nu(s), s \in S$.

[1] Although POMDPs are a highly interesting topic, we will focus only on fully observable processes which will be denoted as MDPs from now on, unless confusion may arise from this notation.

[2] In stage $t = 0$ instead of a transition from a previous state to the current the process just chooses the current state according to the prior probability distribution ν.

[3] Note that both rewards and costs are bounded; treatment of unbounded rewards/costs is not covered in this research.

One can easily see the correspondence between the process description given above and Mary's situation. The day Mary starts the experiment she observes her parents, and their mood sets the current state. Every day that follows is another stage for the process. According to her knowledge of parents' behavior dynamics, the transition probabilities are set as shown in Table 2.1 for action ask_mother, Table 2.2 for action ask_father, and Table 2.3 for action $ask\_neither$. Similarly, the actual numbers corresponding to Mary's observed "reward" function is also shown in Table 2.4. In these tables x_i represents a current states while x_i' represents the state at the next stage.

2.2 Policies

For a Markov decision process, one also needs a well defined mechanism for choosing actions. This mechanism is provided by a *policy*.[4] A policy π is a mapping from states to actions, i.e. for every state, the policy produces an action. If the policy depends on the stage the process is in, then the policy is called non-stationary. Policies can be deterministic, $\pi : S \rightarrow A$, in which case for any given state the action chosen is certain, or stochastic, $\pi : S \times A \rightarrow [0, 1]$, and then a probability distribution is defined over the available actions. Thus, in the latter case, for any given state, an action is chosen with the given probability. In this dissertation, we will focus only on stationary, deterministic policies.

Returning to our simple example, one possible policy—although not very good—for Mary would be to always choose action $ask\_neither$, no matter what mood her parents are in.

We note that fixing the policy reduces the process to a *Markov chain* whose transition probabilities are not functions of the action taken: $P(s_t|s_{t-1})$. In Mary's case, if the policy were fixed to the action $ask\_neither$ in every state, then the transition probability function would consist of only the numbers given in Table 2.3.

2.3 Performance Quantification

Now that we introduced the model, we will turn our attention to how we can quantify a system's performance. Three different objectives are typically used to assess the goodness

[4]The policy is also sometimes referred to as *strategy* or *universal plan*

	x'_1	x'_2	x'_3	x'_4
x_1	0.40	0.60	0.00	0.00
x_2	0.04	0.36	0.06	0.54
x_3	0.08	0.12	0.32	0.48
x_4	0.01	0.09	0.09	0.81

Table 2.1: Probability transition table for Example 2.1.1, action ask_mother.

	x'_1	x'_2	x'_3	x'_4
x_1	0.30	0.70	0.00	0.00
x_2	0.01	0.59	0.00	0.40
x_3	0.06	0.14	0.24	0.56
x_4	0.00	0.10	0.01	0.89

Table 2.2: Probability transition table for Example 2.1.1, action ask_father.

	x'_1	x'_2	x'_3	x'_4
x_1	0.16	0.64	0.04	0.16
x_2	0.00	0.40	0.01	0.59
x_3	0.02	0.08	0.18	0.72
x_4	0.00	0.01	0.01	0.98

Table 2.3: Probability transition table for Example 2.1.1, action $ask\_neither$.

	ask_mother	ask_father	$ask\_neither$
x_1	\$0.50	\$0.00	\$0.00
x_2	\$1.50	\$0.00	\$0.00
x_3	\$1.50	\$2.00	\$0.50
x_4	\$2.50	\$2.00	\$0.50

Table 2.4: Reward function for Example 2.1.1.

of a given policy. The general idea is that one would like to maximize long term reward,[5] but the following refinement is standard in MDP literature: expected total reward, expected average per step reward, and expected total discounted reward. Each of these objectives determines a value associated with every state of the MDP, $V(s)$, commonly called the *value function* ($V : S \rightarrow \mathbb{R}$).

Expected total reward is associated with MDPs of *finite horizon* in which the system's performance is measured over a finite number of stages T:

$$V^{\pi}(s_0) = E_{\pi}\left\{\sum_{t=1}^{T} R(s_t)\right\} \tag{2.2}$$

It is interpreted as the expected sum of rewards if the process starts in some state s_0 and makes T transitions according to some policy π.

Expected average reward per stage is a measure of performance for MDPs making an infinite number of stage transitions, commonly known as *infinite horizon* MDPs. The value function for such processes is defined as:

$$V^{\pi}(s_0) = \lim_{T\rightarrow\infty} E_{\pi}\left\{\frac{1}{T}\sum_{t=1}^{T} R(s_t)\right\}. \tag{2.3}$$

There are problems in which recent rewards are preferable to rewards received in some distant future, an effect achievable by discounting future rewards exponentially. This second case of infinite horizon has a value function of the form:

$$V^{\pi}(s_0) = \lim_{T\rightarrow\infty} E_{\pi}\left\{\sum_{t=1}^{T} \gamma^{t-1} R(s_t)\right\} \tag{2.4}$$

where the discount factor γ takes values between 0 and 1.

To illustrate the three different objectives, we will refer again to Example 2.1.1. If Mary desired to optimize the first objective, she would have to set a horizon value, say of 30 days. Then she would judge a policy by the expected value of the money accumulated within 30 days from the start. However, since for her is more important to receive money presently (or in the immediate future) rather than in some distant future, and since she does not have the stringent requirement of a cutoff horizon, she decides that the third objective suits her best.

[5] To facilitate a simpler presentation the rest of the document will avoid discussing cost, which does not invalidate any of the results presented; the change required to include action costs is trivial.

Thus she chooses to judge any of her policies by the expected accumulated money, where each day she discounts the money's value exponentially.

To clarify notation we re-state that when we refer to the *value of policy* π we mean the value of each state $s \in S$, when, starting in s the process follows policy π for the preset number of stages and accumulates reward according to one of the three schemes just described.

2.4 Optimal Performance

In previous sections we introduced policies, and, to be able to say that one is better than another, we also introduced the concept of value functions. However, the ultimate goal in a problem associated with the MDP is to maximize its performance criterion (e.g. expected discounted reward). In other words we want to find the best or *optimal policy*, denoted by π^*. This policy yields optimal behavior by definition, i.e., if a process starts in any initial state and chooses actions dictated by π^* it will achieve maximum possible expected reward. This is equivalent to saying that no other policy has a better value function and can be written as:

$$V^*(s) = V^{\pi^*}(s) \geq V^{\pi}(s) \quad \forall \pi, s \tag{2.5}$$

where V^{π^*} is the optimal value function, or the value function of the optimal policy, also denoted by V^*. We will tacitly assume that all conditions required for the optimal value function to exist are met, and mention if this is not the case.

When the optimal policy does not exist, we may instead want to find the ϵ-optimal policy, denoted by π^*_ϵ, which satisfies the following condition for all $s \in S$ and some $\epsilon > 0$:

$$V^{\pi^*_\epsilon}(s) \geq V^*(s) - \epsilon. \tag{2.6}$$

Bellman developed a recursive definition of the value function much used in devising various algorithms [2]. If state-values were known at the next stage then calculating the current state's value involves only an expected value computation:

$$V^{\pi}(s_t) = R(s_t, a_t) + \gamma \sum_{s_{t+1}} P(s_{t+1}|s_t, \pi(s_t)) V^{\pi}(s_{t+1}) \tag{2.7}$$

which can be written for all states concurrently using matrix notation:

$$\boldsymbol{V^{\pi}} = \boldsymbol{R^{\pi}} + \gamma \boldsymbol{P^{\pi}} \boldsymbol{V^{\pi}} \tag{2.8}$$

This relation applies to the optimal value function as well, leading to what is known as Bellman's *principle of optimality*:

$$\boldsymbol{V}^* = \max_a \{\boldsymbol{R}^a + \gamma \boldsymbol{P}^a \boldsymbol{V}^*\} \tag{2.9}$$

where the $\max$ operator applies row-wise. For notational convenience we introduce the notion of *greedy policy* with respect to some value function $\boldsymbol{V}$ as

$$\begin{aligned} \pi^{\text{gre}}(V, s_t) &= \arg\max_{a_t \in A(s_t)} \left\{ R(s_t, a_t) + \gamma \sum_{s_{t+1}} P(s_{t+1}|s_t, a_t) V(s_{t+1}) \right\} \\ &= \arg\max_{a_t \in A(s_t)} \{Q(s_t, a_t)\}, \end{aligned} \tag{2.10}$$

or in matrix notation:

$$\begin{aligned} \boldsymbol{\pi}^{\text{gre}}(\boldsymbol{V}) &= \arg\max_a \{\boldsymbol{R}^a + \gamma \boldsymbol{P}^a \boldsymbol{V}\} \\ &= \arg\max_a \{\boldsymbol{Q}^a\}. \end{aligned} \tag{2.11}$$

In other words, this is the policy choosing the action which, if taken, would result in the maximum expected value in the next stage. The state-action value function, $Q^\pi(s_t, a_t)$, is an analog of the value function, namely the expected value obtained by taking action a_t in state s_t and then following policy π until stage T is reached[6][7] ($Q : S \times A \to \mathbb{R}$).

2.5 Problem Types, Solutions, and Nomenclature

Although the main objective of an MDP problem is to find a policy which maximizes some notion of long term reward accumulation, technically one can have additional refinements of the general problem. As an intermediate step to finding the optimal policy, one may instead find the optimal value or state-value function. In other cases, we may be interested in another intermediate result which is finding the value of a given policy. Therefore, depending on what is known about the MDP and what the goal is, different algorithms may apply. The model of the environment, determined in entirety by the transition probabilities P and

[6] Stage T could be infinity.

[7] The un-discounted case differs only by the removal of γ, the discount factor, from Equations 2.7, 2.9, and 2.10.

reward function R, may or may not be known to the solver. If P and R are given, the problem is usually referred to as *solving the MDP*. Problems where a model of the environment is not given fall in the area of *reinforcement learning*. Solutions which explicitly represent either P, or R, or their approximation, are referred to as *model based* or indirect methods. Otherwise we will call them direct methods.

2.6 Computational Complexity of Flat MDP Problems

We add a very brief summary of results pertaining to the hardness of solving MDP problems with a flat representation, in order to motivate further the search for efficient approximate algorithms.

A beginning study in the computational complexity of MDP problems with a flat representation has been studied by Papadimitriou and Tsitsikilis. Their main result is as follows:

Theorem 2.6.1 *[46] The Markov Decision Problem is P-complete in all three cases (finite horizon, discounted, and average cost).*

Intuitively, Theorem 2.6.1 says that the MDP problem—regardless of the optimality criterion chosen—is solvable in polynomial time in the worst case, however, it is unlikely to be able to obtain a significant speed-up by using parallel computation. The complexity class P contains those problems which admit a polynomial time solution using a sequential algorithm. It is known that some of these problems can be solved in a logarithmic-polynomial time[8] using a polynomial number of processors and a parallel algorithm. When this is the case, the problem is said to belong to the class NC. Similar to the question P $=^?$ NP, it is an open question whether NC $=^?$ P (most researchers are inclined to think not). That is, it is unlikely that all sequential problems in P can be parallelized to obtain orders of magnitude speed-ups. The class P-complete is the class containing the hardest problems in P, and, if a parallel algorithm was found that worked for one of these problems, then we would be able to solve all of them efficiently, as transforming one problem into another is guaranteed in a polynomial number of steps. One basic problem that is P-complete is the *circuit value problem* (CVP): given a circuit with boolean inputs and output, and *and or* gates, is its value *true*? The proof of Theorem 2.6.1 involves a reduction from CVP to the MDP problem.

[8] $O(log(n)^k)$, for input of size n using k processors.

Interestingly, linear programming belongs to the P-complete class as well, and thus, the only known polynomial algorithm for solving MDP problems is for the linear programming (LP) formulation given in the following section. The two best classes of algorithms for solving linear programs are the *simplex*-like [13] algorithms and *interior point* methods. In particular, Khachian [32] is thought to be the first to prove that linear programming's complexity is polynomial, while Karmakar [31] being the first to propose a polynomial algorithm with better practical performance. Alas, Karmakar's algorithm is still quite inefficient on many problems, and various implementations of the simplex algorithm are used instead for their excellent average performance, in spite of their worst case exponential complexity. Though, there is room for further good news since it is not known that linear programs resulting from MDP problems may actually exhibit the features needed to make simplex run in exponential time. So far the only problems on which simplex would be inefficient have been highly contrived examples [33].

2.7 Methods for Solving Flat MDPs

Having introduced the problem, now it is time to turn attention to algorithms for solving it. These algorithms historically will fall within three broad categories of techniques which have been the focus of most research pertaining to exact solutions to MDPs: *value iteration*, *policy iteration*, and *linear programming*. Note that these methods have the important property that they are capable of producing the exact solution—as opposed to an approximation—and they are guaranteed to terminate in a finite number of steps.[9] Their input is a *flat* MDP, i.e. the probability transitions and reward function have a description of size polynomial in the number of states.

Although our primary interest is in approximate algorithms, it would be difficult to appreciate the origin, structure, and methodology of the approximation should we choose to overlook exact algorithms. Hence we will provide a concise review of value iteration, policy iteration, and linear programming in the next section, followed by our treatment of approximate algorithms.

Value iteration—shown in Algorithm 1—is part of a larger family of methods, namely *dynamic programming* [30, 6, 4]. It starts with a guess for the value function and updates it iteratively until changes in the value function are smaller than some desired tolerance,

[9] For finite precision inputs.

an event which triggers its termination. There are theoretical guarantees of convergence in a finite number of steps to an ϵ-optimal solution when tables are used to store the value function, reward function, and probability transition function. The value and reward function tables must be able to store a value for each state, while the probability transition function table needs to store a value for each triple of state, next stage state, and action. Therefore the memory requirements of value iteration are polynomial in the state space size.

Algorithm 1 Value Iteration

1: $\boldsymbol{v}^{(0)} \leftarrow \boldsymbol{g},$ set $\epsilon > 0, n \leftarrow 0$
2: **repeat**
3: $\quad n \leftarrow n + 1$
4: $\quad \boldsymbol{v}^{(n)} \leftarrow \max_a \left\{ \boldsymbol{R}^a + \gamma \boldsymbol{P}^a \boldsymbol{v}^{(n-1)} \right\}$
5: **until** $||\boldsymbol{v}^{(n)} - \boldsymbol{v}^{(n-1)}|| < \epsilon$
6: $\boldsymbol{\pi}^*_\epsilon \leftarrow \boldsymbol{\pi}^{\text{gre}}(\boldsymbol{v}^{(n)})$

where $\boldsymbol{g}$ is a vector of guess values, and we switched to matrix notation such that instead of $P(s'|s, a)$ we write $\boldsymbol{P}^a$.

Another popular method, policy iteration, described in Algorithm 2 and introduced by Howard [30], differs from value iteration in that it manipulates a policy explicitly, alternating between a policy evaluation and a policy improvement step until convergence occurs. Although, like value iteration, it most commonly applies to stationary infinite-horizon problems, it can also be applied to more general models under appropriate assumptions. In MDPs with finite states and actions the algorithm finishes with certainty in a finite number of steps. Theoretically the rate of convergence of policy iteration is at least linear [51], with many reporting that in practice the algorithm usually converges at a faster rate than value iteration.

Linear programming is the third exact method covered in this section and is shown in Algorithm 3 [16, 7, 19, 20, 29, 42]. It is not quite as popular and widely used as value or policy iteration is, but its usefulness will be motivated by the following sections, where approximate solutions are sought.

We briefly note that Step 2 of Algorithm 3 can be skipped since the dual solution of the linear program holds the optimal policy.

Although the use of linear programming has been associated with computational inefficiency in the past, much progress has been made in this area. In practice, large linear programs can currently be solved in very reasonable times, giving this method a new level

Algorithm 2 Policy Iteration

1: $\boldsymbol{\pi}^{(0)} \leftarrow \boldsymbol{h}, n \leftarrow 0$
2: **repeat**
3: $\quad n \leftarrow n + 1$
4: $\quad$ (Policy evaluation)
$\quad \boldsymbol{v}^{(n)} \leftarrow$ solution of linear system $\boldsymbol{v}^{(n)} = \boldsymbol{R}^a + \gamma \boldsymbol{P}^{\boldsymbol{\pi}^{(n-1)}} \boldsymbol{v}^{(n)}$
5: $\quad$ (Policy improvement)
$\quad \boldsymbol{\pi}^{(n)} \leftarrow \boldsymbol{\pi}^{\text{gre}}(\boldsymbol{v}^{(n)})$
6: **until** $\boldsymbol{\pi}^{(n)} = \boldsymbol{\pi}^{(n-1)}$
7: $\boldsymbol{\pi}^* \leftarrow \boldsymbol{\pi}^{(n)}$

where $\boldsymbol{v}^{(n)}$ is a vector of unknowns, and $\boldsymbol{h}$ is a vector holding an arbitrary policy.

of credibility. When solving the entire linear program at once memory usage is potentially higher than that of other methods, but can be overcome using constraint generation. Trick and Zin [58] demonstrate the viability of using linear programming specifically for solving MDPs.

Algorithm 3 Linear Programming

1: Solve the linear optimization:
$\quad \boldsymbol{V}^* = \arg\min_{\boldsymbol{v}} \boldsymbol{v}^T \mathbf{1}$
$\quad$ subject to
$\quad \boldsymbol{v} \geq \boldsymbol{R}^a + \gamma \boldsymbol{P}^a \boldsymbol{v} \quad \forall a$
2: $\boldsymbol{\pi}^* \leftarrow \boldsymbol{\pi}^{\text{gre}}(\boldsymbol{V}^*)$

where the parameter vector $\boldsymbol{v}$ is unrestricted and the objective function is an inner product between $\boldsymbol{v}$ transpose and a vector of ones.

2.8 Structured MDPs and Compact Representation

Our interest in approximate algorithms stems from the fact that non-concisely represented MDPs will lead exact algorithms to provide solutions in very impractical times. Therefore, we will allow MDPs to be represented somehow compactly; however, the discerning reader may ask: how useful could this be if the actual solution does not have a compact representation?—not very useful indeed. Therefore, to keep our hope of developing practical methods

intact, we allow the solution to be represented succinctly as well. As soon as this allowance is made, it should become clear very quickly that we might have to give up exact optimal solutions and instead settle for some sort of good approximations, and we will expand later on what "good" means for us.

At first sight, the results of Section 2.6 seem acceptable; after all, a polynomial run time is not undesirable; upon closer examination one notices that the run time is polynomial in the size of the state space, which, for large MDPs, can be exponential in the number of variables representing the state, as we will see later in this section. As expected, real life problems are notorious for their large size, and, if one hopes to employ MDPs for modeling such problems, then the need for more efficient solutions arises. A way out of this impediment may be based on the fact that real life problems often exhibit structure mostly overlooked by general algorithms such as those just described. Common sense dictates that significant structure in a problem can be used to one's advantage. Of course, this may not always be the case, but nevertheless, when it is true, we would like to have methods which exploit structure. The search for new algorithms and heuristics is also warranted by the inherent hardness of the problem regardless of whether an exact or an approximate solution is desired [39, 43, 23, 22, 40, 41] (Details in Section 2.9.)

The type of structure we are interested in is that which makes it possible to represent large parts of the system by a succinct equivalent, of logarithmic order of the size of the original. Furthermore we would like to find algorithms which:

- have a small memory footprint, when compared to the size of the state space,
- give the user the ability to trade off accuracy for run time,
- and if stopped anytime, a current solution is available.

2.8.1 Factored State Space and Dynamics

The first components whose representation size we address are the state space and the process dynamics (i.e. probability transition function and reward function).

The state space Instead of indexing the entire state space with integers, we assume that certain features of interest can be identified in the problem at hand. Features can be represented using propositional variables, allowing one to obtain a one-to-one correspondence

between configurations of said variables and states. Before describing this formally, let us illustrate the practice with a modeling exercise of Example 2.1.1.

Recall the set of states $\{x_1, x_2, x_3, x_4\}$ of Example 2.1.1, representing the moods of Mary's parents. Instead, we can use a binary variable for each parent:

- $F \in \{0, 1\}$ represents father's mood. Variable F takes value 1 if father is in a good mood, and 0 otherwise.
- $M \in \{0, 1\}$ represents mother's mood. Variable M takes value 1 if mother is in a good mood, and 0 otherwise.

Thus the set of all configurations of the ordered pair (F, M) is used as the set of states, with the understanding that the vector $\boldsymbol{x} = (M, F)$.

We generalize from this example and assume [10] that the state space is encoded via a vector $\boldsymbol{x}$ of n binary *state variables*, $x_1, \ldots, x_n$. The state space size is then $N = 2^n$, with each state being an assignment of true or false values to the n variables. State spaces described in this way are sometimes called *factored*. Similarly, we will represent action $\boldsymbol{a}$ by k binary action variables $a_1, \ldots, a_k$.

The probability transition function The dynamics of a temporally changing probabilistic system can be represented compactly in several ways, but each representation has its own advantages and disadvantages, and not all are fitting efficient algorithms.

Conceptually speaking, we need to be able to represent the probability of arriving in some state knowing that some action was taken in the present state, and that we have a finite number of states and actions. With our choice of using a set of propositional variables to encode the state space, we have committed to the idea that the probability transition function is a mapping from the set of configurations of state variables to the interval $[0, 1]$. Therefore, $P : Dom(\boldsymbol{x}') \times Dom(\boldsymbol{x}) \times Dom(\boldsymbol{a}) \rightarrow [0, 1]$. We already introduced the simplest representation of this mapping as a table, and now survey other more compact possibilities. It is true that for some problems a table can be sparse, i.e. have in fact few non-zero entries, and therefore have a small memory requirement, but this is clearly not always the case. Furthermore, the sparsity patterns can be highly different from a problem to another, and algorithms may not be able to always take advantage of such structure. We

[10] Without loss of generality in the case of finite state spaces.

would like to find representation classes such that there exists certain structure in a given class that allows an algorithm to be efficient on all the class members.

We are motivated to choose the *dynamic Bayes network* (DBN) [17] representation both for its expressive power and for its suitability to the task we undertake. A number of researchers have used this representation successfully in solving MDP related problems, and we cite the work of Koller, Parr, and Guestrin as the most relevant work to the current research [34, 35, 27, 28].

A DBN is a Bayes network with an extension enabling it to represent certain temporal information. Let us define a Bayes network first.

Definition 2.8.1 A Bayes network is a joint probability distribution defined on a finite set $\boldsymbol{X} = \{x_1, \ldots, x_n\}$ of discrete variables as a product of conditional probability functions:

$$P(\boldsymbol{X}) = \prod_{i=1}^{n} P(x_i|\text{pa}(x_i))$$

where $\text{pa}(x_i) \subseteq \boldsymbol{X}$. (The set $\text{pa}(x)$ is often called the *parents of* x.) Furthermore, for any $x_i, i = 1 \ldots n$, if $A_0 = \{x_i\}$, $A_j = \cup_{y \in A_{j-1}} \text{pa}(y)$, and $A_j \neq \emptyset$, $j = 1, 2, \ldots$, then $A_0 \cap A_j = \emptyset$.[11]

Another—less formal—way of describing a Bayes net is graphical, due to the one-to-one correspondence that exists between Bayes networks and directed acyclic graphs. To construct a graph for a Bayes network one lets each variable $x \in \boldsymbol{X}$ be a graph vertex, and draws directed edges from vertex y to vertex x if $y \in \text{pa}(x)$. A directed link between two vertices means that the two variables are probabilistically dependent. Figure 2.1 illustrates the concept. For a more extensive treatment of Bayes nets and associated algorithms we recommend Pearl's book [48].

A dynamic Bayes net aims at representing graphically a joint probability function whose variables are associated with different time stages. In MDPs, leveraging the Markov property—future states depend only on the current state—enables us to discuss only two stage DBNs (e.g. like that shown in Figure 2.2).

Dependencies between next stage and current stage state variables are problem dependent—they encompass one aspect of what we call structure. If the number of actions is small one can choose not to represent actions via propositional variables and, instead, use

[11] That is, one cannot arrive at the same node while being allowed to move only from nodes to parents.

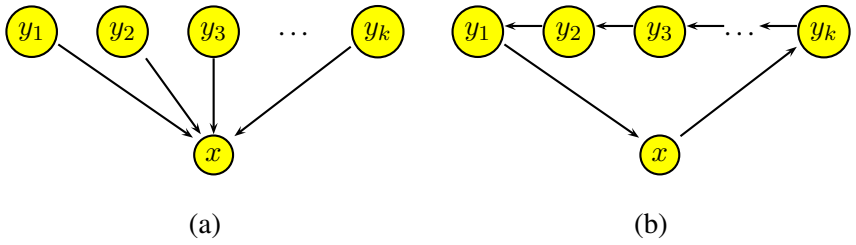


Figure 2.1: (a) A Bayes net: graphical representation of the joint probability function $P(x, y_1, \ldots, y_k) = P(x|y_1, \ldots, y_k) \prod_{i=1}^{k} P(y_i)$; (b) not a Bayes net.

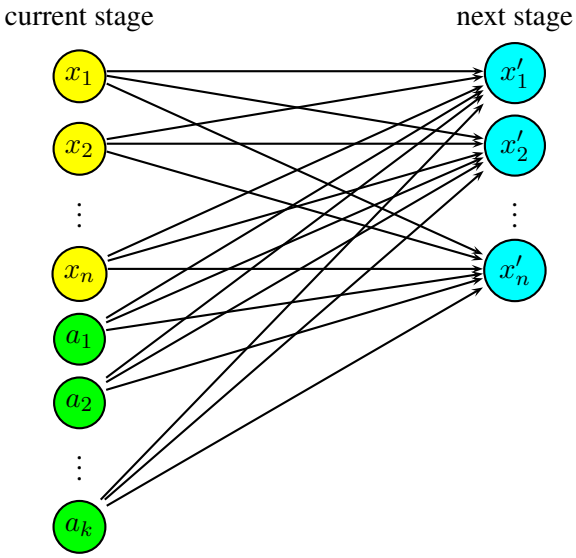


Figure 2.2: Dynamic Bayes net example: propositional state variables in "next stage" depend only on propositional state and action variables in "current stage."

one DBN for each action. How small the number of actions should be depends on the available memory and computational power.

Thus the DNB in this figure represents a joint probability distribution over variables appearing in two time stages:[12]

$$P(\boldsymbol{x}'|\boldsymbol{a},\boldsymbol{x}) = P(x'_1, x'_2, \ldots, x'_n|a_1, a_2, \ldots, a_k, x_1, x_2, \ldots, x_n) \quad (2.12)$$

$$= \prod_{i=1}^{n} P(x'_i|a_1, a_2, \ldots, a_k, \text{pa}(x'_i)) \quad (2.13)$$

where $\text{pa}(x'_i)$ are parent state variables with links pointing to x'_i.

We note that the terms participating in the product expression of the joint are commonly called factors. Normally each factor is represented as a table of values: one for each configuration of its variables.[13] If one prefers a more compact representation, for example because the factors happen to be functions of many variables, one can choose a succinct function representation such as *decision trees* [10].

The reward function Since reward is a function of state and action, we assume it factorizes additively over subsets of state and action variables [1]. Formally, given a collection $\{\boldsymbol{x}\}_j$, $j = 1, 2, \ldots, m$, and $m << 2^n$, such that each $\boldsymbol{x}_j \subseteq \boldsymbol{x}$ is a subset of all state variables, we assume that

$$R(\boldsymbol{x}, \boldsymbol{a}) = \sum_{j=1}^{m} R(\boldsymbol{x}_j, \boldsymbol{a}).$$

This assumption is not very restrictive and quite natural for many decomposable domains where each component brings its contribution to the total reward. One extreme case is when the summation has only one term which is a function of all the state and action variables.

2.8.2 Compact Value Function Representation

For large MDPs, the explicit representation of the value function as a table is out of the question. Candidate compact representations include state to value mappings, state to features mappings, or state to feature to value mappings. Examples of these are neural networks [5], decision trees [10], algebraic decision diagrams [56], linear combinations of basis functions

[12] We will consistently use primed variables to denote that they belong to the "next stage."

[13] Usually referred to as *conditional probability table* or CPT.

[54, 34, 35, 53, 47], piecewise linear combinations (decision tree with linear combinations of basis functions at leaves) [50], and state aggregation [55]. Although the main requirement on a candidate representation is that it is compact, from a computational efficiency standpoint the different representations may not suit known algorithms well. There are still open questions on how much a computational saving (if any) one can obtain using compact representations for specific problems. In general, of course, mere compactness does not guarantee faster and/or more precise algorithms.

An important issue in non-explicit value function representation is that of approximation. If the function exhibits structure which the compact representation fits exactly, then no approximation is needed. For example this might be the case if the value function lies in the space spanned by a given basis. However, this happens only in very simple instances and almost never in realistic domains, even if the rest of the model is factored and compactly represented in the best possible way. Therefore, it is most likely that the true value function will need to be approximated by some function belonging to a restricted family, and this is the view that we adopt here. The restricted family we use is that of linear combinations of *basis functions*. Although other compact representations have been proposed elsewhere with various degrees of success, we can now further research with a linear MDP approximation architecture.

Consider a function $V : S \rightarrow \mathbb{R}$ whose domain is a set of states (configurations of state variables), such as described in the previous section. In our case this function is the MDP value function, but this would apply to any function defined on a number of multi-valued variables. Instead of a flat representation, i.e. a table for holding a real value for each state, we will use a number of basis[14] functions:

$$\begin{aligned} &\phi_1 : S \rightarrow \mathbb{R} \\ &\phi_2 : S \rightarrow \mathbb{R} \\ &\vdots \\ &\phi_K : S \rightarrow \mathbb{R} \end{aligned}$$

where each member of S is a configuration of state variables $x_1, \ldots, x_n$. However, to keep each function easily computable we would like to restrict each ϕ_i to a small subset of state variables $\boldsymbol{x}_i$. For instance ϕ_1's domain $\boldsymbol{x}_1$ could be (x_3, x_5, x_n), and, if variables x_3, x_5, and x_n were binary, a possible choice for the basis function could be *logic and*, i.e. $\phi_1 =$

[14] We use this name although the functions need not necessarily constitute an algebraic basis.

$x_1 \wedge x_5 \wedge x_n$. Note that if we allow the basis functions to have unlimited domain sizes, one can simply go to the extreme of using all the state variables, which would defeat our purpose of having a compact representation.

The goal is to find parameter (weights) vector $\boldsymbol{w}$ such that

$$\hat{V}(\boldsymbol{x}) = \sum_{i=0}^{K} w_i \phi_i(\boldsymbol{x})$$

is a close approximation of $V(\boldsymbol{x})$, for any $\boldsymbol{x}$. The notion of "close" is defined as a small distance between the approximation and the optimal value function. In theory one could choose any one of several distance functions, such as:

$$\begin{array}{lcl} L_1 & = & ||V - \hat{V}||_1 = \sum_s |V(s) - \hat{V}(s)|, \\ L_{1,c} & = & ||V - \hat{V}||_{1,c} = \sum_s |V(s) - \hat{V}(s)|\, c(s), \\ L_2^2 & = & ||V - \hat{V}||_2^2 = \sum_s (V(s) - \hat{V}(s))^2, \text{ or} \\ L_\infty & = & ||V - \hat{V}||_\infty = \max_s |V(s) - \hat{V}(s)|, \end{array}$$

but the actual choice can make quite a big difference in practice; for instance, minimizing the L_1 or weighted L_1 distance is easier than minimizing the L_∞ distance, as it will be seen in Section 2.9.

To simplify notation somewhat we can represent the K basis functions as the matrix they induce, i.e.

$$\boldsymbol{\Phi} = \begin{bmatrix} \boldsymbol{\phi_1}(\boldsymbol{x}_1) & \boldsymbol{\phi_2}(\boldsymbol{x}_1) & \cdots & \boldsymbol{\phi_K}(\boldsymbol{x}_1) \\ \vdots & \vdots & \vdots & \vdots \\ \boldsymbol{\phi_1}(\boldsymbol{x}_N) & \boldsymbol{\phi_2}(\boldsymbol{x}_N) & \cdots & \boldsymbol{\phi_K}(\boldsymbol{x}_N) \end{bmatrix}$$

Thus $\hat{\boldsymbol{V}} = \boldsymbol{\Phi w}$. We will describe how we make use of this representation in the next chapters.

2.9 Computational Complexity of Succinctly Represented MDP problems

Before we engage in designing algorithms, it would be good to know what we are up against; i.e., how difficult is the task of solving MDPs if we indeed are given a compact representation. We review below some important results.

From a computational complexity point of view, a compact representation such as described above is equivalent to the *sequential-effects tree* (ST) representation introduced by Littman [38]. In ST representation the state variables are propositions (in our case boolean), and the effect of each action on a proposition is encoded in a decision tree instead of a table. The leaves of the tree give the values to which the proposition will change as an action's effect. The leaves therefore encode a *true* or *false* value, with a given probability. Since the details of the representation are not of crucial importance, the reader is directed to Littman's paper for more explanations. The following results uses this representation.

Theorem 2.9.1 *[38] The plan-existence problem for ST is EXPTIME-complete.*

Essentially, this theorem tells us that succinctly represented MDPs are hard problems. For completeness we will elaborate a little on the class EXPTIME-complete, but for more background on the subject we recommend Papadimitriou's book [44]. This complexity class is thought of as including the hardest problems in the set EXPTIME. The complexity class EXPTIME is made of problems which are solvable by a deterministic Turing machine in $O(2^{p(n)})$ time, where $p(n)$ is a polynomial function of input size n. In our case, n would be the size of the succinct MDP description. Two explanations are in order: first, Littman presents this result in a planning context, but it applies directly to MDPs; second, complexity results are usually presented for a decision problem, whereas solving an MDP is clearly an optimization problem. The work around given by all practitioners is that any optimization problem can be transformed into a binary search using decision problems. For instance, instead of finding the optimal policy, we can ask "is there a policy with value greater than δ?" for various values of δ.

The other interesting question, regarding the computational complexity of policy evaluation in compactly represented MDPs is— as far as we know—an open question [43].

As we can see, simply having a smaller representation does not make the problem easy in general. This does not mean that we should give up trying to find specific problem domains for which such representation can help; rather it reinforces what experience has shown again and again, that real life problems which rarely are unstructured, are very hard to solve.

2.10 Approximate Algorithms for Solving MDPs

To obtain approximate algorithms from the exact version of policy iteration and linear programming one can replace each $V(\boldsymbol{x})$ variable with the summation $\sum_{i=1}^{K} w_i \phi_i(\boldsymbol{x})$, for which the basis functions ϕ_i are known, and a new set of variables $\boldsymbol{w}$ is introduced. Thus reducing the problem dimension from N to K variables. This modification has been shown and analyzed by Schweitzer and Seidmann [54]. Their approximate policy iteration is shown in Algorithm 4.

Algorithm 4 Approximate Policy Iteration – Schweitzer and Seidmann (API-SS)

1: $\boldsymbol{\pi}^{(0)} \leftarrow \boldsymbol{h}, n \leftarrow 0, n_{\text{MAX}}$
2: **repeat**
3: $\quad n \leftarrow n + 1$
4: $\quad$ (Policy evaluation)
$\quad \boldsymbol{w}^{(n)} \leftarrow \arg\min_{\boldsymbol{w}} \{[\boldsymbol{R} + \gamma \boldsymbol{P}^{\boldsymbol{\pi}^{(n-1)}} \boldsymbol{\Phi}\boldsymbol{w} - \boldsymbol{\Phi}\boldsymbol{w}]^2 \boldsymbol{c}\}$
5: $\quad$ (Policy improvement)
$\quad \boldsymbol{\pi}^{(n)} \leftarrow \boldsymbol{\pi}^{\text{gre}}(\boldsymbol{\Phi}\boldsymbol{w})$
6: **until** $\boldsymbol{\pi}^{(n)} = \boldsymbol{\pi}^{(n-1)}$ or $n \leq n_{\text{MAX}}$
7: **if** $n \leq n_{\text{MAX}}$ **then**
8: $\quad \boldsymbol{\pi}^{*} \leftarrow \boldsymbol{\pi}^{(n)}$
9: **else if** $\boldsymbol{\pi}^{(n)} \neq \boldsymbol{\pi}^{(n-1)}$ **then**
10: $\quad$ return UNSUCCESSFUL

where $\boldsymbol{h}$ is a vector holding an arbitrary policy, n_{MAX} is the maximum number of iterations allowed, and $\boldsymbol{c}$ is a vector of positive weights.

The n_{MAX} limit on the number of iterations must be imposed to prevent cycling because, unlike the exact version of policy iteration, the sequence of value functions is not necessarily monotone. Note that in the policy evaluation step, the necessary optimization evaluation amounts to a regular least squares problem minimization, hence, zeroing the gradient of the right hand side with respect to $\boldsymbol{w}$ leads to a system of K linear equations in unknowns $\boldsymbol{w}$.

Koller and Parr provided an approximate method for policy evaluation using ideas similar to those of Schweitzer and Seidmann. They argued [34] that the preferred instantiation of weights $\boldsymbol{c}$ is $\boldsymbol{\rho}$, the stationary distribution of the underlying Markov process. However, since $\boldsymbol{\rho}$ is not known *a priori*, they use an approximation and derive an iterative approximate

dynamic programming algorithm. However, using the stationary distribution in minimizing the weighted Euclidean distance was a principal hindrance in further development of a subroutine to policy iteration. Hence, Koller and Parr address this problem [35] and improve the algorithm by replacing the stationary distribution with arbitrary non-negative weights, and replacing the iterative dynamic programming with a direct way of evaluating the policy, thus offering an approximate policy iteration algorithm. It should be noted that Koller and Parr addressed an important issue which Schweitzer and Seidmann treatment lacked, regarding the use of a compact policy. Under somewhat restrictive assumptions they represented the working policy as a *decision list*

$$\langle \boldsymbol{x}_1, a_1, \delta(\boldsymbol{x}_1, a_1)\rangle, \langle \boldsymbol{x}_2, a_2, \delta(\boldsymbol{x}_2, a_2)\rangle, \ldots \langle \boldsymbol{x}_L, a_L, \delta(\boldsymbol{x}_L, a_L)\rangle \tag{2.14}$$

where $\boldsymbol{x}_l, l = 1, \ldots, L$, is an assignment to a subset of variables in the state vector $\boldsymbol{x}$, effectively using a pattern of partial assignments to state variables to represent subsets of states to which the same action applies. In order to understand the decision list representation one needs to look first at the required assumptions. The model must have a default action d which leads to the concept of a default state-action value function $\boldsymbol{Q}^d$. A score $\delta(s, a)$ is defined as the difference between the value of the given state-action pair and that of the default state-action pair: $\delta(s, a) = Q(s, a) - Q(s, d)$. The decision list is sorted in decreasing order of δ. Thus, a possibly compact mapping from states to actions is provided since for any state $\boldsymbol{x}$, the first triple $\langle \boldsymbol{x}_l, a_l, \delta(\boldsymbol{x}_l, a_l)\rangle$ whose pattern $\boldsymbol{x}_l$ is a match provides the action a_l with maximum Q value, thus the greedy action is easily obtained.

Guestrin *et al.* [27, 28] introduced a further improvement over Koller and Parr's algorithm. The least-squares minimization was discarded in favor of the L_∞ criterion:

$$\boldsymbol{w}^{(\boldsymbol{n})} \leftarrow \arg\min_{\boldsymbol{w}} \left\| \boldsymbol{R} + \left(\gamma \boldsymbol{P}^{\boldsymbol{\pi}^{(n-1)}} - \boldsymbol{I}\right) \boldsymbol{\Phi}\boldsymbol{w} \right\|_\infty \tag{2.15}$$

which can be solved via the linear program:

$$\begin{aligned} &\min_{\boldsymbol{w}, y} y \\ &\text{subject to} \\ &\quad y\,\mathbf{1} \;\geq\; \boldsymbol{R}^{\boldsymbol{\pi}} + \left(\gamma \boldsymbol{P}^{\boldsymbol{\pi}} - \boldsymbol{I}\right) \boldsymbol{\Phi}\boldsymbol{w} \\ &\quad y\,\mathbf{1} \;\geq\; \left(\boldsymbol{I} - \gamma \boldsymbol{P}^{\boldsymbol{\pi}}\right) \boldsymbol{\Phi}\boldsymbol{w} - \boldsymbol{R}^{\boldsymbol{\pi}} \end{aligned} \tag{2.16}$$

for some policy $\boldsymbol{\pi}$, represented as a decision list. This change facilitated the computation of error bounds on the algorithm's solution.

2.10.1 Linear Basis Functions and the Exploitation of Structure

This section deals with the practicality of these approximate algorithms and it describes material developed by Koller, Parr, and Guestrin. The linear program of Equation 2.16 is a good example of the computational issues needed to be tackled. Not only does the linear program feature the matrix multiplication $\boldsymbol{P}^{\boldsymbol{\pi}}_{\boldsymbol{x},:}\boldsymbol{\Phi}\boldsymbol{w}$ for each constraint (sizes $1 \times N, N \times K$, and $K \times 1$ respectively), but it also has $2N$ number of constraints; for very large N this is a linear program of significant size. We use $\boldsymbol{P}^{\boldsymbol{\pi}}_{\boldsymbol{x},:}$ to denote the row vector of matrix $\boldsymbol{P}^{\boldsymbol{\pi}}$ corresponding to state $\boldsymbol{x}$. Fortunately, specific structure in the MDP, such as described earlier, allows the matrix product computation to be done in an efficient manner. For this to hold, besides the default action assumption, a number of other conditions must be met:

- each basis function depends only on few state variables,
- the dynamic Bayes net representing the transition probability distribution has *next state* variables with few parent *current state* variables (see Figure 2.3),

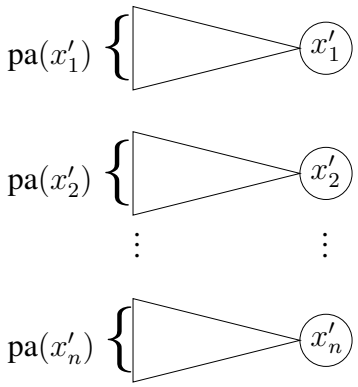


Figure 2.3: Each state variable $x'_1, x'_2, \ldots, x'_n$ in the next stage has few parent variables $\text{pa}(x_i)$ in the current stage.

- the reward function decomposes additively and each factor is a function of few state variables.

The following derivation shows how helpful these assumptions really are in reducing the computational effort:

$$\begin{aligned}
\boldsymbol{P}^{\boldsymbol{\pi}}_{\boldsymbol{x},:}\boldsymbol{\Phi}\boldsymbol{w} &= \sum_{\boldsymbol{x}'} P(\boldsymbol{x}'|\boldsymbol{x},\pi(\boldsymbol{x}))\sum_{i=1}^{K} w_i\phi_i(\boldsymbol{x}') \\
&= \sum_{i=1}^{K} w_i \sum_{\boldsymbol{x}'} P(\boldsymbol{x}'|\boldsymbol{x},\pi(\boldsymbol{x}))\phi_i(\boldsymbol{x}') \\
&= \sum_{i=1}^{K} w_i \sum_{\boldsymbol{x}'_i} P(\boldsymbol{x}'_i|\boldsymbol{x},\pi(\boldsymbol{x}))\phi_i(\boldsymbol{x}'_i) \sum_{\boldsymbol{x}'_{\boldsymbol{j}}\in\boldsymbol{x}'\setminus\boldsymbol{x}'_i} P(\boldsymbol{x}'_{\boldsymbol{j}}|\boldsymbol{x},\pi(\boldsymbol{x})) \\
&= \sum_{i=1}^{K} w_i \sum_{\boldsymbol{x}'_i} P(\boldsymbol{x}'_i|\boldsymbol{x},\pi(\boldsymbol{x}))\phi_i(\boldsymbol{x}'_i) \qquad (2.17)
\end{aligned}$$

After moving the summation over basis functions out to the left in Step 2, we divide the summation over all the variables into two summations, one over variables $\boldsymbol{x}'_i$, which is the subset of state variables in $\boldsymbol{x}'$ that the basis function ϕ_i depends on, and the other over variables $\boldsymbol{x}'_{\boldsymbol{j}}$, which is the disjoint subset of $\boldsymbol{x}'$ on which the basis function ϕ_i does not depend on. This summation over all values of variables occurring in the probability transition function adds up to one, and therefore does not appear in the last step. Thus, the $2^{|\boldsymbol{x}'|} = 2^n$ summation over variables $\boldsymbol{x}'$ has been reduced to a summation with $2^{|\boldsymbol{x}'_i|}$ terms, no longer a computational challenge for basis functions of small number of variables.

Without further changes, the computation in Equation 2.17 needs to be done for each of the $2N$ constraints. To overcome this difficulty Guestrin *et al.* employ a trick inspired from the technique of *variable elimination* [18, 3]. First the constraints must be written in a more amenable form.[15] Instead of

$$y\,\mathbf{1} \geq \boldsymbol{R}^{\boldsymbol{\pi}} + (\gamma\boldsymbol{P}^{\boldsymbol{\pi}} - \boldsymbol{I})\,\boldsymbol{\Phi}\boldsymbol{w} \qquad (2.18)$$

or, in non-matrix form

$$y \geq R(\boldsymbol{x},\pi(\boldsymbol{x})) + \gamma\sum_{\boldsymbol{x}'} P(\boldsymbol{x}'|\boldsymbol{x},\pi(\boldsymbol{x}))\sum_{i=1}^{K} w_i\phi_i(\boldsymbol{x}') - \sum_{i=1}^{K} w_i\phi_i(\boldsymbol{x}) \qquad \forall\boldsymbol{x}\in S$$

$$R(\boldsymbol{x},\pi(\boldsymbol{x})) + \gamma\sum_{\boldsymbol{x}'} P(\boldsymbol{x}'|\boldsymbol{x},\pi(\boldsymbol{x}))\sum_{i=1}^{K} w_i\phi_i(\boldsymbol{x}') - \sum_{i=1}^{K} w_i\phi_i(\boldsymbol{x}) \leq y \qquad \forall\boldsymbol{x}\in S,$$

[15] We show it only for $y\,\mathbf{1} \geq \boldsymbol{R}^{\boldsymbol{\pi}} + (\gamma\boldsymbol{P}^{\boldsymbol{\pi}} - \boldsymbol{I})\,\boldsymbol{\Phi}\boldsymbol{w}$, as the transformation for $y\,\mathbf{1} \geq (\boldsymbol{I} - \gamma\boldsymbol{P}^{\boldsymbol{\pi}})\,\boldsymbol{\Phi}\boldsymbol{w} - \boldsymbol{R}^{\boldsymbol{\pi}}$ is identical except for some sign changes.

we can write

$$\max_{\boldsymbol{x}\in S}\left\{R(\boldsymbol{x},\pi(\boldsymbol{x}))+\gamma\sum_{\boldsymbol{x}'}P(\boldsymbol{x}'|\boldsymbol{x},\pi(\boldsymbol{x}))\sum_{i=1}^{K}w_i\phi_i(\boldsymbol{x}')-\sum_{i=1}^{K}w_i\phi_i(\boldsymbol{x})\right\}\leq y. \quad (2.19)$$

At this point having a decision list representation and a small number of actions become crucial assumptions. Recall that the decision list has the desirable property of an inverse mapping: i.e. can produce a set of patterns of assignments to state variables for any given action (in effect it partitions the state space according to the action the policy would choose from those states). Allow $\pi_{\text{inv}} : A \Rightarrow \mathcal{P}(S)$ to denote the inverse mapping from actions to subsets of S. Therefore $\pi_{\text{inv}}(a)$ would be the set of states policy π would choose action a from. Thus, for each $a \in A$ we have a constraint similar to Equation 2.19:

$$\max_{\boldsymbol{x}\in\pi_{\text{inv}}(a)}\left\{R(\boldsymbol{x},\pi(\boldsymbol{x}))+\gamma\sum_{\boldsymbol{x}'}P(\boldsymbol{x}'|\boldsymbol{x},a)\sum_{i=0}^{K}w_i\phi_i(\boldsymbol{x}')-\sum_{i=0}^{K}w_i\phi_i(\boldsymbol{x})\right\}\leq y. \quad (2.20)$$

The significance of the transformation will become clear once we do some more mathematical manipulation of this last equation:

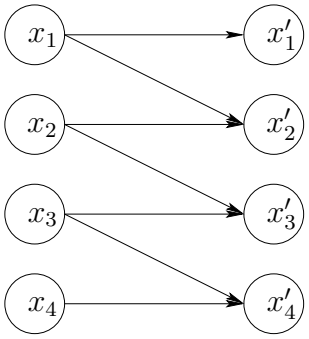
$$\begin{aligned}\max_{\boldsymbol{x}\in\pi_{\text{inv}}(a)}\left\{R(\boldsymbol{x},\pi(\boldsymbol{x}))+\gamma\sum_{\boldsymbol{x}'}P(\boldsymbol{x}'|\boldsymbol{x},a)\sum_{i=0}^{K}w_i\phi_i(\boldsymbol{x}')-\sum_{i=0}^{K}w_i\phi_i(\boldsymbol{x})\right\} &\leq y\\ \max_{\boldsymbol{x}\in\pi_{\text{inv}}(a)}\left\{\sum_{j=1}^{m}R(\boldsymbol{x}_j,\pi(\boldsymbol{x}))+\sum_{i=0}^{K}w_i\gamma\sum_{\boldsymbol{x}'_i}P(\boldsymbol{x}'_i|\mathbf{pa}(\boldsymbol{x}'_i),a)\phi_i(\boldsymbol{x}'_i)-\phi_i(\boldsymbol{x}_i)\right\} &\leq y.\end{aligned} \quad (2.21)$$

Note that the reward is explicitly decomposed into m sub-reward functions, each a function of only $\boldsymbol{x}_j \subseteq \boldsymbol{x}$ state variables. We also introduce the following notation:

$$\mathbf{pa}(\boldsymbol{x}'_i)=\bigcup_{k,x'_k\in\boldsymbol{x}'_i}\text{pa}(x'_k).$$

In general, the max in Equation 2.21 can be over the entire state space and therefore costly to calculate. Besides that, the max operator also makes the constraints non-linear. Guestrin *et al.* employ a clever trick to transform constraints of this type back into linear constraints at the cost of introducing extra variables. For simplicity it would be best to illustrate this with an example (here the reward only depends on state). Assume we have the following MDP:

- current stage states are $\boldsymbol{x} = (x_1, x_2, x_3, x_4)$ and next stage states are $\boldsymbol{x}' = (x'_1, x'_2, x'_3, x'_4)$ where states variables take binary values,
- reward decomposes as $R(\boldsymbol{x}, \pi(\boldsymbol{x})) = \sum_{j=1}^{4} R(x_j, \pi(\boldsymbol{x}))$,
- system dynamics for action a according to the DBN in the following figure:



  correspond to

$$P(\boldsymbol{x}'|\boldsymbol{x}, a) = P(x'_1|x_1)P(x'_2|x_2, x_1)P(x'_3|x_3, x_2)P(x'_4|x_4, x_3),$$

- $\hat{V}(\boldsymbol{x}) = \sum_{i=1}^{4} w_i \phi_i(x_i)$
- and policy π chooses the same action a in all states, or $\pi_{\text{inv}}(a) = S$.

Re-writing Equation 2.21 for this example we obtain

$$\begin{aligned}
\max_{x_1} \max_{x_2} \ldots \max_{x_4} \quad \{ \quad & R(x_1) + R(x_2) + R(x_3) + R(x_4) + \\
& w_1 \gamma \sum_{x'_1} P(x'_1|x_1, a)\phi(x'_1) - \phi(x_1) + \\
& w_2 \gamma \sum_{x'_2} P(x'_2|x_2, x_1, a)\phi(x'_2) - \phi(x_2) + \\
& w_3 \gamma \sum_{x'_3} P(x'_3|x_3, x_2, a)\phi(x'_3) - \phi(x_3) + \\
& w_4 \gamma \sum_{x'_4} P(x'_4|x_4, x_3, a)\phi(x'_4) - \phi(x_4) \quad \} \leq y
\end{aligned}$$

and, after moving each $\max$ operator inside as far as it can go,

$$\begin{aligned}
\max_{x_1} \quad & \{ \quad R(x_1) + w_1\gamma \sum_{x_1'} P(x_1'|x_1, a)\phi(x_1') - \phi(x_1) + \\
\max_{x_2} \quad & \{ \quad R(x_2) + w_2\gamma \sum_{x_2'} P(x_2'|x_2, x_1, a)\phi(x_2') - \phi(x_2) + \\
\max_{x_3} \quad & \{ \quad R(x_3) + w_3\gamma \sum_{x_3'} P(x_3'|x_3, x_2, a)\phi(x_3') - \phi(x_3) + \\
\max_{x_4} \quad & \{ \quad R(x_4) + w_4\gamma \sum_{x_4'} P(x_4'|x_4, x_3, a)\phi(x_4') - \phi(x_4) \quad \}\}\}\} \leq y
\end{aligned} \tag{2.22}$$

The construction used by Guestrin *et al.* is based on variable elimination. In general, the variable elimination algorithm maintains a collection F of functions, initially instantiated to $\{f_1, f_2, \ldots, f_m\}$, the functions present in the formula to be maximized/minimized. From the still to be eliminated variables a variable x_l is chosen for elimination. The set of functions $G = \{f \in F : x_l \in \mathbf{Dom}(f)\}$ is replaced with a new function e_l whose domain is $\cup_{g \in G}\mathbf{Dom}(g) \backslash \{x_l\}$. For our example, assume that variable x_4 is chosen to be eliminated in Equation 2.22; then function $e_4(x_3)$ replaces $\max_{x_4}\{R(x_4)+w_4\gamma \sum_{x_4'} P(x_4'|x_4, x_3, a)\phi(x_4') - \phi(x_4)\}$. This procedure is repeated until all variables have been eliminated and the result of the maximization/minimization is returned. For our example this is equivalent to a constant replacing the left hand side of the constraint of Equation 2.22. The cost of variable elimination is shown to be exponential [18] in the induced tree width obtained during the variable elimination procedure. Intuitively, the induced tree width is the size of the largest domain of replacing functions e_l (minimized over variable orderings).

Back to the constraints construction; eliminating the first variable x_1: for each instantiation y_1 of variables in $\boldsymbol{y}_1 = \mathbf{Dom}(e_1)$, a new LP variable $u_{y_1}^{e_1}$ and associated constraints are introduced. Continuing with the example, when eliminating x_4, two variables $u_0^{e_4}$ and u_1^{e4}, and two new constraints $u_0^{e4} = e_4(0)$ and $u_1^{e4} = e_4(1)$ are introduced, corresponding to instantiating x_3 in $e_4(x_3)$ to 0 and 1 respectively. For each subsequent eliminated variable x_l new LP variables are introduced for each instantiation of variables in the domain of the function to be replaced, and a separate constraint for each as well. Assuming we are

eliminating x_3 next, we introduce $u_0^{e_3}$ and $u_1^{e_3}$ and constraints

$$\begin{aligned}
u_0^{e_3} &\geq R(x_3=0) + w_3\gamma \sum_{x_3'} P(x_3'|x_3=0, x_2=0, a)\phi(x_3') - \phi(x_3=0) + u_0^{e_4} \\
u_0^{e_3} &\geq R(x_3=1) + w_3\gamma \sum_{x_3'} P(x_3'|x_3=1, x_2=0, a)\phi(x_3') - \phi(x_3=1) + u_1^{e_4} \\
u_1^{e_3} &\geq R(x_3=0) + w_3\gamma \sum_{x_3'} P(x_3'|x_3=0, x_2=1, a)\phi(x_3') - \phi(x_3=0) + u_0^{e_4} \\
u_1^{e_3} &\geq R(x_3=1) + w_3\gamma \sum_{x_3'} P(x_3'|x_3=1, x_2=1, a)\phi(x_3') - \phi(x_3=1) + u_1^{e_4}
\end{aligned}$$

After eliminating the last variable and introducing new LP variables and constraints, a few last additional constraints are still needed to make the connection to the original constraint. For instance, after eliminating x_1, we still need:

$$\begin{aligned}
u &\geq u_0^{e_1} \\
u &\geq u_1^{e_1} \\
u &\leq y.
\end{aligned}$$

Here variable u takes the value of the max expression on the left hand side of Equation 2.22.

One should note that this procedure is provably correct, all new constraints are linear in both the new LP variables and the original LP variables $\boldsymbol{w}$, and the LP transformed this way is equivalent to the original LP (i.e. the two LPs have the same optimal solution $\boldsymbol{w}$) [27]. Thus this construction offers the advantage of a compact LP, where instead of N constraints there are $O(2^k|A|)$ constraints, and k is the size of the largest domain over all replaced functions during the transformation.

2.10.2 Approximate Policy Iteration Using a Compact LP

Calculating the value of a policy efficiently encourages one to use the policy iteration algorithm. Guestrin *et al.* introduced approximate versions of both *value iteration* and *policy iteration* with a preference for the latter, as it seemed to converge faster in practice. We, for the same reason, will also maintain a focus on policy iteration. Their algorithm differs from that of Schweitzer and Seidmann (Algorithm 4) in that the policy evaluation step is

performed using the compact LP described in the previous section (a compact version of Equation 2.16).

Chapter 3

Approximation Error

It is important to recall that only in rare cases the optimal value function lies in the set spanned by the given basis functions. Therefore, one should expect that $\boldsymbol{\Phi w}$ will only be an approximation. There are two questions of interest: first, what is the quality of the approximate value function induced by $\boldsymbol{w}$—regardless of how this approximation was obtained, and second, can the error (or a bound on the error) be calculated efficiently?

The chapter is laid out as follows: Section 3.1 describes two results related to the difficulty of computing the Bellman error. Section 3.2 introduces a branch-and-bound algorithm which can calculate the Bellman error, while Section 3.3 illustrates its use with some experiments.

In regards to the first question, all we have is an upper bound shown by Williams and Baird [59]. This bound is defined in terms of the *Bellman error*; for any value function $\boldsymbol{V}$ the Bellman error is defined as:

$$\text{BellErr}(\boldsymbol{V}) = \| \max_a \{\boldsymbol{R}^a + \gamma \boldsymbol{P}^a \boldsymbol{V}\} - \boldsymbol{V} \|_\infty, \tag{3.1}$$

or the largest absolute value discrepancy between the value function of any state and the value obtained by passing $\boldsymbol{V}$ through the Bellman *back-up*.[1] One can use the Bellman error to obtain a crude upper bound on the L_∞ error of the approximate value function $\boldsymbol{\Phi w}$:

$$\|V^* - \boldsymbol{\Phi w}\|_\infty \leq \frac{\text{BellErr}(\boldsymbol{\Phi w})}{1-\gamma}. \tag{3.2}$$

As can be seen from Equation 3.2, to obtain this bound one must be able to compute the Bellman error. The next section explores how difficult a task this is.

3.1 Hardness of Error Calculation

We show two results—due to Patrascu *et al.* [47]—related to this computation: one that claims that it is hard to assess whether the Bellman error is above or below a given threshold, and therefore to compute it; and another results that claims that it is hard to determine whether a current approximation can be improved with respect to the Bellman error.

Theorem 3.1.1 *It is co-NP-complete to determine whether the Bellman error of a given $\boldsymbol{\Phi w}$ is less than a given δ.*

[1] $\boldsymbol{R}^a + \gamma \boldsymbol{P}^a \boldsymbol{V}$.

Proof. We show that the complementary problem of deciding whether the Bellman error is at least δ is NP-complete. To show that it is in NP: a given witness state s can be used to certify a large Bellman error, and this can be done in polynomial time using the structured computation:

$$\max_a |(\boldsymbol{\Phi}\boldsymbol{w})(s) - \gamma \sum_{s'} P(s'|s,a)(\boldsymbol{\Phi}\boldsymbol{w})(s')|.$$

To show the problem is NP-hard we use a reduction from 3SAT: given a 3CNF formula, let the state variables correspond to propositional (formula) variables. Construct a basis function ϕ_j for each clause, such that it indicates whether the clause is satisfied by the state assignment. Set the rewards to zero and the transition model to identity for each action, and set $\gamma = 0$ and $\boldsymbol{w} = \mathbf{1}$. The Bellman error for this setup becomes $\max_s \sum_{j=1}^{k} \phi_j(\boldsymbol{s}_j)$. If k is the number of clauses, then the Bellman error will be k if and only if the original 3CNF formula is satisfiable. ■

Theorem 3.1.2 *It is NP-hard to determine whether there exists a weight vector $\boldsymbol{w}$ such that $\boldsymbol{\Phi}\boldsymbol{w}$ has Bellman error less than a given δ. The problem remains in NP$^{\text{co-NP}}$.*

Proof. First, to establish that the problem is in NP$^{\text{co-NP}}$, note that an acceptable $\boldsymbol{w}$ can be given as a certificate of small Bellman error, and this can then be verified by consulting a co-NP oracle. Second, NP-hardness follows from a reduction from 3SAT: given a 3CNF formula, let the state variables correspond to the propositional variables, and construct a local reward function r_j for each clause that is the same for each action, where r_j is the indicator function for satisfaction of clause j. Choose a single trivial basis function $\phi_0 = 0$. Set the transition model to be identity for each action and set $\gamma = 0$. The Bellman error in this setup becomes $\max_s \sum_{j=1}^{k} r_j(\boldsymbol{s}_j)$. If k is the number of clauses, then $\min_w \max_s \sum_{j=1}^{k} r_j(\boldsymbol{s}_j)$ yields value k if and only if the original 3CNF formula is satisfiable. ■

3.2 An Algorithm to Compute the Bellman Error

In the light of these two results one should expect that, in general, calculating the Bellman error of a given solution $\boldsymbol{w}$ is not a practical proposition. However, we propose a scheme which in many cases, for MDPs that are not extremely large, will be much more practical than a brute force approach. For ease of presentation we introduce some notation. Let

$$
\begin{aligned}
v(\boldsymbol{w}, \boldsymbol{x}) &\triangleq (\boldsymbol{\Phi}\boldsymbol{w})(s), \\
q(\boldsymbol{w}, \boldsymbol{x}, a) &\triangleq r(\boldsymbol{x}, a) + \gamma \sum_{\boldsymbol{x}'} P(\boldsymbol{x}'|\boldsymbol{x}, a) v(\boldsymbol{w}, \boldsymbol{x}').
\end{aligned}
$$

We would like to calculate $\max_{\boldsymbol{x}} |v(\boldsymbol{w}, \boldsymbol{x}) - \max_a q(\boldsymbol{w}, \boldsymbol{x}, a)|$, which can be reduced to two searches:

$$
\begin{aligned}
&\min_{\boldsymbol{x}} \quad v(\boldsymbol{w}, \boldsymbol{x}) - \max_a q(\boldsymbol{w}, \boldsymbol{x}, a) \\
&\max_{\boldsymbol{x}} \quad v(\boldsymbol{w}, \boldsymbol{x}) - \max_a q(\boldsymbol{w}, \boldsymbol{x}, a) \qquad (3.3) \\
& \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad (3.4)
\end{aligned}
$$

The first search is easy, since

$$
\begin{aligned}
\min_{\boldsymbol{x}} v(\boldsymbol{w}, \boldsymbol{x}) - \max_a q(\boldsymbol{w}, \boldsymbol{x}, a) &= \min_{\boldsymbol{x}} \min_a v(\boldsymbol{w}, \boldsymbol{x}) - q(\boldsymbol{w}, \boldsymbol{x}, a) \\
&= \min_a \min_{\boldsymbol{x}} v(\boldsymbol{w}, \boldsymbol{x}) - q(\boldsymbol{w}, \boldsymbol{x}, a) \qquad (3.5)
\end{aligned}
$$

Here, once a is fixed, the min over $\boldsymbol{x}$ can be efficiently computed by using variable elimination the usual way. The second search is much harder.

$$
\max_{\boldsymbol{x}} v(\boldsymbol{w}, \boldsymbol{x}) - \max_a q(\boldsymbol{w}, \boldsymbol{x}, a) = \max_{\boldsymbol{x}} \min_a v(\boldsymbol{w}, \boldsymbol{x}) - q(\boldsymbol{w}, \boldsymbol{x}, a) \qquad (3.6)
$$

The problem in this case is that we can no longer use simple variable elimination to conduct the search, because a cannot be held fixed while eliminating $\boldsymbol{x}$. However, we can perform this search by using a branch and bound method.

First note that Equation 3.6 can be broken down into a number of constrained maximum searches, one for each a, where $\boldsymbol{x}$ is restricted to those states where in fact a is the optimal action.

$$
\max_{\boldsymbol{x}} v(\boldsymbol{w}, \boldsymbol{x}) - \max_a q(\boldsymbol{w}, \boldsymbol{x}, a) = \max_a \max_{\boldsymbol{x}: \boldsymbol{x} \in \pi_{\boldsymbol{w}}^{-1}(a)} v(\boldsymbol{w}, \boldsymbol{x}) - q(\boldsymbol{w}, \boldsymbol{x}, a) \qquad (3.7)
$$

where $\pi_{\boldsymbol{w}}^{-1}$ is the *inverse* of policy $\pi_{\boldsymbol{w}}$, and therefore $\boldsymbol{x} : \boldsymbol{x} \in \pi_{\boldsymbol{w}}^{-1}(a)$ denotes the subset of states that have a as the action chosen by policy $\pi_{\boldsymbol{w}}$. Thus, instead of searching over the whole state space, we only need search over this restricted subset. Also,

$$
\pi_{\boldsymbol{w}}(\boldsymbol{x}) = \arg\max_a q(\boldsymbol{w}, \boldsymbol{x}, a)
$$

is the implicit greedy policy defined by $\boldsymbol{w}$ for $v(\boldsymbol{w}, \cdot)$. Now consider the search for some fixed a_1

$$\begin{aligned} \max_{\boldsymbol{x}:\boldsymbol{x}\in\pi_{\boldsymbol{w}}^{-1}(a_1)} v(\boldsymbol{w},\boldsymbol{x}) - q(\boldsymbol{w},\boldsymbol{x},a_1) \quad \equiv \quad & \max_{\boldsymbol{x}} v(\boldsymbol{w},\boldsymbol{x}) - q(\boldsymbol{w},\boldsymbol{x},a_1) \\ & \text{subject to} \quad q(\boldsymbol{w},\boldsymbol{x},a_1) \geq q(\boldsymbol{w},\boldsymbol{x},a_2) \\ & \qquad\qquad \vdots \\ & \qquad\qquad q(\boldsymbol{w},\boldsymbol{x},a_1) \geq q(\boldsymbol{w},\boldsymbol{x},a_\ell) \end{aligned} \tag{3.8}$$

We solve this with a branch and bound [45] search on state variables in $\boldsymbol{x}$, as shown in Algorithm 5, which is a general branch-and-bound procedure modified to fit our requirements.

Algorithm 5 Branch-and-Bound Search

```
activeset ← {⟨[], [x_1, ..., x_n]⟩}, lowerbound ← +∞, currentbest ← anything
while activeset not empty do
    choose tuple_k = ⟨[a_1 ... a_k], [x_{k+1} ... x_n]⟩ ∈ activeset
    activeset ← activeset − {tuple_k}
    for a_{k+1} ∈ Dom(x_{k+1}) do
        upperbound ← upper_bound([a_1 ... a_{k+1}], [x_{k+2} ... x_n])
        if upperbound ≥ lowerbound and k + 1 = n then
            lowerbound ← upperbound
            currentbest ← [a_1 ... a_{k+1}]
        else if upperbound ≥ lowerbound then
            activeset ← activeset ∪ {⟨[a_1 ... a_{k+1}], [x_{k+2} ... x_n]⟩}
return currentbest and lowerbound
```

where the activeset variable is the current set of tuples of partial assignments to state variables (a_i), and the upper_bound() procedure calculates the upper bound using the Lagrangian as described below.

To calculate upper bounds, consider the Lagrangian of Equation 3.8

$$\begin{aligned} L(\boldsymbol{x},\boldsymbol{\mu}) \quad = \quad v(\boldsymbol{w},\boldsymbol{x}) - q(\boldsymbol{w},\boldsymbol{x},a_1) \; & + \mu_2[q(\boldsymbol{w},\boldsymbol{x},a_1) - q(\boldsymbol{w},\boldsymbol{x},a_2)] \\ & \vdots \\ & + \mu_\ell[q(\boldsymbol{w},\boldsymbol{x},a_1) - q(\boldsymbol{w},\boldsymbol{x},a_\ell)] \end{aligned} \tag{3.9}$$

For any $\boldsymbol{\mu} \geq 0$ we know that maximizing Equation 3.9 over $\boldsymbol{x}$ gives an upper bound on Equation 3.8.

$$\begin{aligned}
&\max_{\boldsymbol{x}} v(\boldsymbol{w},\boldsymbol{x}) - q(\boldsymbol{w},\boldsymbol{x},a_1) \; + \mu_2[q(\boldsymbol{w},\boldsymbol{x},a_1) - q(\boldsymbol{w},\boldsymbol{x},a_2)] \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \vdots \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad + \mu_\ell[q(\boldsymbol{w},\boldsymbol{x},a_1) - q(\boldsymbol{w},\boldsymbol{x},a_\ell)] \\
\geq \quad &\max_{\boldsymbol{x}:\boldsymbol{x}\in\pi_{\boldsymbol{w}}^{-1}(a_1)} v(\boldsymbol{w},\boldsymbol{x}) - q(\boldsymbol{w},\boldsymbol{x},a_1) \; + \mu_2[q(\boldsymbol{w},\boldsymbol{x},a_1) - q(\boldsymbol{w},\boldsymbol{x},a_2)] \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \vdots \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad + \mu_\ell[q(\boldsymbol{w},\boldsymbol{x},a_1) - q(\boldsymbol{w},\boldsymbol{x},a_\ell)] \\
\geq \quad &\max_{\boldsymbol{x}:\boldsymbol{x}\in\pi_{\boldsymbol{w}}^{-1}(a_1)} v(\boldsymbol{w},\boldsymbol{x}) - q(\boldsymbol{w},\boldsymbol{x},a) \quad \text{for } \boldsymbol{\mu} \geq 0
\end{aligned}$$

Therefore, any $\boldsymbol{\mu} \geq 0$ gives us an efficient way of calculating an upper bound on Equation 3.8 since $\max_{\boldsymbol{x}} L(\boldsymbol{x},\boldsymbol{\mu})$ can be efficiently calculated using variable elimination.

This upper bound can actually be improved by updating $\boldsymbol{\mu}$ in a negative subgradient direction. That is, starting with $\boldsymbol{\mu}^0$, let $\boldsymbol{x}^0 = \arg\min_{\boldsymbol{x}} L(\boldsymbol{x},\boldsymbol{\mu}^0)$. Then a subgradient direction is given by

$$\widetilde{\nabla}_{\boldsymbol{\mu}} L(\boldsymbol{x}^0,\boldsymbol{\mu}^0) \;=\; \begin{bmatrix} q(\boldsymbol{w},\boldsymbol{x}^0,a_1) - q(\boldsymbol{w},\boldsymbol{x}^0,a_2) \\ \vdots \\ q(\boldsymbol{w},\boldsymbol{x}^0,a_1) - q(\boldsymbol{w},\boldsymbol{x}^0,a_\ell) \end{bmatrix}$$

Thus, a tighter upper bound can be obtained by updating $\boldsymbol{\mu}$ in the negative subgradient direction

$$\boldsymbol{\mu}^{t+1} \;=\; \boldsymbol{\mu}^t - \alpha\widetilde{\nabla}_{\boldsymbol{\mu}} L(\boldsymbol{x}^t,\boldsymbol{\mu}^t), \quad \text{for } \alpha > 0$$

Since we also want to maintain the constraint that $\boldsymbol{\mu} \geq 0$ (and for other reasons) we prefer the exponentiated subgradient update

$$\boldsymbol{\mu}^{t+1} \;=\; \boldsymbol{\mu}^t \beta^{-\widetilde{\nabla}_{\boldsymbol{\mu}} L(\boldsymbol{x}^t,\boldsymbol{\mu}^t)}, \quad \text{for } \beta > 1$$

Overall, this suggests that reasonable upper bounds on Equation 3.8 can be calculated after a few alternating rounds of variable elimination (maximizing over $\boldsymbol{x}$) and subgradient update steps. It is critical to obtain good upper bounds on Equation 3.8 because the quality of these bounds strongly determines the efficiency of a branch and bound search over $\boldsymbol{x}$.

3.3 Experiments—Calculating the Bellman Error

3.3.1 Motivation, Goals, and Setup

Just from the fact that this algorithm performs a branch-and-bound search and knowing its theoretical limitations we can conclude that one can always find a problem that will force the method to spend an inordinate computation time. However, sometimes, in practice, one can observe much better behavior and, lacking theory to predict it, one is left only with the choice of performing an empirical evaluation. This is our case as well.

We test the practicality of the branch-and-bound algorithm by running experiments on the `cycle` and `3legs` problems, with a discount factor of 0.95. The two problems are suitable for this type of experimentation due to their ease of scalability and understanding. We describe the two problems in in detail in Appendix A. The code was written in Matlab and run on 750MHz Pentium III level machines.

We set up the simulation as follows. The solution for each problem is approximated by the weighted sum of basis functions including the constant 1 function and bit identity functions (which we call singletons because they take as argument the value of only one state variable). Since we would like to be able to calculate the Bellman error particularly of solutions from our algorithms, we obtain the weights using our approximate linear programming with constraint generation algorithm[2] that we cover in detail in Chapter 4.

3.3.2 Results and Discussion

We varied the `cycle` problem's size between 12 and 32 state variables and the `3legs` size between 13 and 28 state variables; the largest instances of these problems featured 4e10 and 3e9 states respectively. Although we are not aware of other algorithms which can compute the Bellman error efficiently for instances of similar size, we do not consider these problems very large. Table 3.1 gives an indication of the computation times needed to calculate the Bellman error. Naturally, the trend in computation time is exponential, but the growth is not monotonic; this can be seen on the `3legs` problem on instances of 22, 25, and 28 state variables, where the branch-and-bound algorithm jumped from 866 seconds to a little over

[2]We kindly ask for the reader's understanding when we pre-maturely mention an algorithm that has yet to be described; although for the general assessment of the branch-and-bound error calculation any approximate solution—vector of finite values $\boldsymbol{w}$—would have sufficed, calculating the Bellman error of solutions obtained using one of our algorithms seems a somewhat better fit to this thesis.

The `cycle` Problem

$n =$	12	15	18	20	24	28	32
$N =$	4e4	3e5	3e6	1e7	2e8	3e9	4e10
Time (s)	63	131	225	279	451	959	1086

The `3legs` Problem

$n =$	13	16	19	22	25	28
$N =$	8e4	7e5	5e6	4e7	3e8	3e9
Time (s)	525	291	3454	866	14639	2971

Table 3.1: Calculating the Bellman error: results using singleton bases.

four hours and then down again to about 50 minutes. The same behavior is not seen on the `cycle` problem whose times grow in a more monotonic fashion. We confess to be in the same predicament as those who use similar branch-and-bound algorithms and are not able to predict on which problem instance the search will behave well or not. In spite of the inherent danger of such practices, if we ventured to draw any conclusion about the frequency of spiked times from the tabulated results, it would be that this does not happen very often. Thus it is to be expected that once in a while, the branch-and-bound computation will take a long time to finish.

Since the exponential trend is unmistakable, these results point out that on very large problems of over 40 state variables the branch-and-bound method will most likely be inadequate. However, for problems smaller than 40 variables but larger than around 30 variables our method remains the more efficient algorithm.

Although the size of the Bellman error is of more interest in Chapter 4 and becomes the principal subject of Chapter 5, we highlight here the trend in error growth versus problem size. Figure 3.1 shows plots of the error for the two problems we experimented on. It is a pleasant surprise to see that the error plots have an almost perfectly linear trend. This gives us hope that the choice of approximating architecture coupled with our optimization method does not deteriorate the solution as the problem size increases; we defer a more detailed look at this to following chapters.

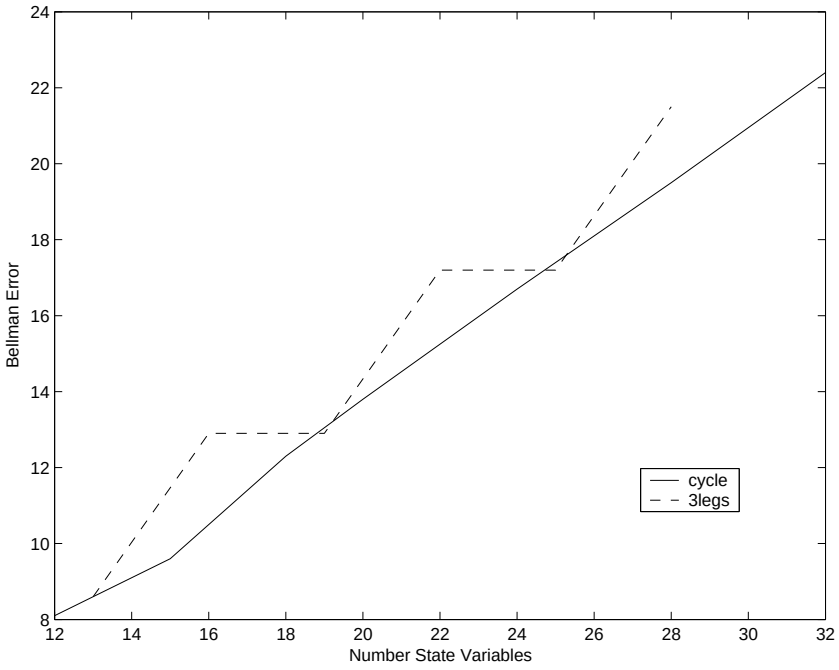


Figure 3.1: The Bellman error calculated using the branch-and-bound method for a sequence of instances of the `cycle` and `3legs` problems.

Chapter 4

Obtaining an MDP Approximate Solution

4.1 Introduction

When it comes to actually solving factored MDPs, the approximate policy iteration of Guestrin *et al.* can be considered state of the art. However, it would be desirable to get rid of the "default action" assumption, and, in the same time to obtain a solution faster. The goal of this chapter is to address these two issues, in the context of a given compactly represented MDP and solution approximated via a linear combination of given basis functions.

The conceptual layout of the chapter is as follows: we start by showing in Section 4.2 the direct method which forms the basis for further solutions we use and then move on to our contribution. The contribution of Section 4.3 is the introduction of two novel algorithms for solving factored MDPs, based on the primal of a linear program. Section 4.4's contribution is the introduction of the branch-and-bound approximate policy iteration, which no longer uses the "default action" assumption by representing the policy implicitly. Section 4.5 investigates the possibility of obtaining an approximate solution via the linear programming dual. While principled, the LP dual based method eventually produces results of poor accuracy and thus, unless positively changed, is not of practical interest to us.

4.2 Approximate Linear Programming

A more direct way to solve an MDP than approximate policy or approximate value iteration has also been introduced by Schweitzer and Seidmann [54]. This method employs linear programming, and is almost identical to Algorithm 3 from Section 2.7 except for using $\hat{\boldsymbol{V}} = \boldsymbol{\Phi}\boldsymbol{w}$ instead of $\boldsymbol{V}$ as the approximating step. Although Schweitzer and Seidmann have not observed it, the algorithm can be used in larger problems if they exhibit the type of structure we are able to cope with and was described above. We will show how this works shortly. The general form of the method is shown in Algorithm 6 below. The LP featured in this algorithm is feasible, provided one of the columns of $\boldsymbol{\Phi}$ is $\mathbf{1}$ [54]. Left in this general form, the LP is practically intractable for large problems. To bring the computation back to the doable realm the following assumptions—presented on page 28 and repeated here for ease of reading—are in order:

- basis functions depend on few state variables,
- the dynamic Bayes net representing the transition probability distribution has *next state* variables with few parent *current state* variables (see Figure 2.3),

Algorithm 6 Approximate Linear Programming – General Form

1: Solve the linear program:

$\min_{\boldsymbol{w}}(\boldsymbol{\Phi w})^T \mathbf{1}$

subject to

$\boldsymbol{\Phi w} \geq \boldsymbol{R}^a + \gamma \boldsymbol{P}^a \boldsymbol{\Phi w} \quad \forall a$

2: return $\boldsymbol{\pi}^{\text{gre}}(\boldsymbol{\Phi w})$

where the parameter vector $\boldsymbol{w}$ is unrestricted.

- the number of actions is small,
- the reward function decomposes additively and each factor is a function of few state variables.

It is important to note that the default action assumption employed by Koller, Parr, and Guestrin is no longer needed for making Algorithm 6 efficient.

Algorithm 6 can be brought to a specific form more suitable for scaling up which takes advantage of the assumptions above. First, the objective function can be reduced to a simpler calculation as follows:

$$\begin{aligned}
(\boldsymbol{\Phi w})^T \mathbf{1} &= \sum_{\boldsymbol{x}} \sum_{i=0}^{K} \phi_i(\boldsymbol{x}) w_i \\
&= \sum_{i=1}^{K} w_i \sum_{\boldsymbol{x}_j : \boldsymbol{x}_j = \boldsymbol{x} - \mathbf{Dom}(\phi_i)} \left(\sum_{\boldsymbol{x}_i : \boldsymbol{x}_i = \mathbf{Dom}(\phi_i)} \phi_i(\boldsymbol{x}_i) \right) \\
&= \sum_{i=1}^{K} w_i \, 2^{n - |\mathbf{Dom}(\phi_i)|} \sum_{\boldsymbol{x}_i : \boldsymbol{x}_i = \mathbf{Dom}(\phi_i)} \phi_i(\boldsymbol{x}_i) \\
&= \left[\varphi_0 \varphi_1 \cdots \varphi_K \right] \boldsymbol{w} \\
&= \boldsymbol{\varphi}^T \boldsymbol{w} \qquad (4.1)
\end{aligned}$$

where $\boldsymbol{x}_i$ is the subset of state variables $\boldsymbol{x}$ which form the domain of basis function ϕ_i, and $\varphi_i = 2^{n - |\mathbf{Dom}(\phi_i)|} \sum_{\boldsymbol{x}_i : \boldsymbol{x}_i = \mathbf{Dom}(\phi_i)} \phi_i(\boldsymbol{x}_i)$.

Note: the second line of Equation 4.1 explicitly wrote the summation over all state variables as two summations, one over those variables which are part of ϕ_i's domain, and the other over those variables which are not; $2^{n - |\mathbf{Dom}(\phi_i)|}$ is the number of "$\sum_{\boldsymbol{x}_i : \boldsymbol{x}_i = \mathbf{Dom}(\phi_i)} \phi_i(\boldsymbol{x}_i)$" terms in the "$\sum_{\boldsymbol{x}_j : \boldsymbol{x}_j = \boldsymbol{x} - \mathbf{Dom}(\phi_i)}$" summation, hence the substitution.

Therefore, the LP objective is represented as a K-dimensional inner product between a vector of coefficients and the LP variables w. Each of these coefficients can be determined prior to solving the LP in time exponential in the number of variables each basis function is defined on. Assuming that each basis function has a small domain makes this calculation fast.

In its general form the linear program of Algorithm 6 has N constraints, which is problematic. There are two ways to bypass this difficulty and solve the LP more efficiently. On the one hand, the constraints can be transformed using an almost identical procedure to that of Guestrin *et al.* (detailed in Section 4.3.1). On the other hand, constraints can be left in their original form, but not used all at once, instead generating constraints as needed [53] (Shown in Section 4.3.2).

4.3 Obtaining an Approximation Using the Primal LP

4.3.1 Approximate Linear Programming Via Compact Constraints

We have already shown how to transform constraints to a more compact form on pages 30–33; the resulting method, shown in Algorithm 7, is a new algorithm, introduced independently by Schuurmans and Patrascu [53] and Guestrin *et al.* [28]. This produces a weight vector $\boldsymbol{w}$ which in turn can be used to obtain an implicit greedy policy. The reason for the policy being implicit is that, since the default action assumption has been abandoned, the policy can no longer be represented compactly using a decision list. However, for any given state $\boldsymbol{x}$ an action can efficiently be obtained thus resulting in a well defined policy (see Algorithm 7).

4.3.2 Approximate Linear Programming With Generated Constraints

The technique of solving large LPs using constraint generation is related to the more general method of *cutting planes* and is well established [13, 4] in the optimization literature. More recently it was used by Trick and Zin [57] specifically to solve large MDPs. They report on several versions of constraint generation; however, not all are applicable here.

Our version of constraint generation sufficiently differs in practice from Algorithm 7 that we describe it here separately. The idea behind Algorithm 8 (and in general constraint generation) is based on the fact that the optimal solution of a linear program with many

Algorithm 7 Approximate Linear Programming – Guestrin *et al.* transformed constraints

1: Solve the linear program:

$$w^* = \underset{[\boldsymbol{w}\ \boldsymbol{u}]^T}{\arg\min} \begin{bmatrix} \boldsymbol{\varphi}^T & \mathbf{0} \end{bmatrix} \begin{bmatrix} \boldsymbol{w} \\ \mathbf{0} \end{bmatrix}$$

subject to

Guestrin *et al.* transformed constraints

2: return $\pi^{\text{gre}}(\hat{V}(\boldsymbol{x}), \boldsymbol{x}) = \pi^{\text{gre}}(\sum_{i=1}^{K} \phi_i(\boldsymbol{x}) w_i^*, \boldsymbol{x})$ for any $\boldsymbol{x} \in S$

where the parameter vector $\boldsymbol{w}$ is unrestricted, and the parameter vector $\boldsymbol{u}$ is obtained using the transformation described earlier.

Algorithm 8 Approximate Linear Programming – Constraints Generation

1: $C \leftarrow \emptyset$, $\boldsymbol{w}^{(0)} \leftarrow \mathbf{0}$, $i \leftarrow 0$, $\epsilon \leftarrow$ small positive real number, $\boldsymbol{\delta} \leftarrow 0$.
2: **repeat**
3: $\quad i \leftarrow i + 1$
4: $\quad \delta \leftarrow \min_{\boldsymbol{x},a} \left\{ \Delta(\boldsymbol{x}, a, \boldsymbol{w}^{(i-1)}) \right\}$
5: $\quad \boldsymbol{x}^a \leftarrow \arg\min_{\boldsymbol{x}} \left\{ \Delta(\boldsymbol{x}, a, \boldsymbol{w}^{(i-1)}) \right\} \quad \forall a$
6: $\quad$ **if** $\delta + \epsilon < 0$ **then**
7: $\quad\quad C \leftarrow C \cup \{c^a : c^a$ is the constraint corresponding to state $\boldsymbol{x}^a\}, \forall a$
8: $\quad\quad$ Solve the linear program:
$\quad\quad \min_{\boldsymbol{w}^{(i)}} \boldsymbol{\varphi}^T \boldsymbol{w}^{(i)}$
$\quad\quad$ subject to constraints C
9: **until** $\delta + \epsilon \geq 0$
10: return $\pi^{\text{gre}}(\hat{V}(\boldsymbol{x}), \boldsymbol{x}) = \pi^{\text{gre}}(\sum_{i=1}^{K} \phi_i(\boldsymbol{x}) w_i, \boldsymbol{x})$ for any $\boldsymbol{x} \in S$

where $\Delta(\boldsymbol{x}, a, \boldsymbol{w}) = w_i \sum_{i=1}^{K} \phi_i(\boldsymbol{x}) w_i - R(\boldsymbol{x}, a) - \gamma \sum_{\boldsymbol{x}'} P(\boldsymbol{x}'|\boldsymbol{x}, a) \sum_{i=1}^{K} \phi_i(\boldsymbol{x})$ and the parameter vector $\boldsymbol{w}$ is unrestricted.

more constraints than variables (or vice versa) depends entirely on a small number of constraints, usually referred to as the *active* or *binding constraints*. Except for maligned cases, the number of active constraints is equal to the number of LP variables, which in our case is K. Therefore the algorithm aims at iteratively building a set of constraints, initially empty, that will include—in a small number of iterations—all active constraints. The choice of constraints to add at each iteration i is motivated by the simplex method of solving linear programming problems; in our case, for each action a the constraint featuring greatest violation with respect to current weights $\boldsymbol{w}^{(i)}$ is found (Steps 4 and 5) and included in the set of current constraints C (Step 7). The algorithm terminates when all constraints are satisfied, a condition necessary and sufficient for the linear programming to be solved (within a given tolerance ϵ).

We already showed that the LP objective coefficients can be calculated efficiently and therefore the algorithm's efficiency depends heavily on how efficient the minimization of Steps 4 and 5 can be performed. Again, using the variable elimination inspired procedure of Guestrin *et al.* [27], this minimization poses no problems subject to the stated assumptions. The search for the state-action pair which minimizes the LP constraint expression

$$w_i \sum_{i=1}^{K} \phi_i(\boldsymbol{x}) w_i - R(\boldsymbol{x}, a) - \gamma \sum_{\boldsymbol{x}'} P(\boldsymbol{x}'|\boldsymbol{x}, a) \sum_{i=1}^{K} \phi_i(\boldsymbol{x})$$

differs from the procedure explained on page 32 in that no variables are introduced to replace each inner $\min$ operation; rather, the $\min$ operation is actually carried out. The difficulty of the search is dictated by the domain sizes of basis functions and the degree to which these domains share state variables.

4.3.3 Experiments with the Primal LP Approximation

Motivation, Goals, and Setup

This section illustrates some of the main features of interest regarding the approximate methods described above. We would like to compare the novel direct LP approximating methods (compact and generated constraints) with the approximate policy iteration of Guestrin et al. The main items we would like to address through these simulations are the quality of an approximate solution and the speed with which it was computed.

We ran simulations on the `cycle` and `3legs` problems with up to 40 variables using a discount factor of 0.95. Detailed description of these two problems—including transition

probabilities and reward function values—is given in Appendix A.1. The code was written in Matlab and we called CPLEX 7.5 routines whenever we needed the LP solver.

As calculating the L_∞ or Bellman error is impractical for such large systems, we measure the approximation error via the upper bound on the Bellman error normalized by the maximum reward Rmax. Recall the Bellman error is:

$$\begin{aligned}
\text{BellErr}(\boldsymbol{\Phi w}) &= \|\boldsymbol{\Phi w} - \max_a \{\boldsymbol{R}^a + \gamma \boldsymbol{P}^a \boldsymbol{\Phi w}\}\|_\infty \\
&= \max_s |\boldsymbol{\Phi w} - \max_a \{\boldsymbol{R}^a + \gamma \boldsymbol{P}^a \boldsymbol{\Phi w}\}| \\
&= \max_s |\min_a \{\boldsymbol{\Phi w} - \boldsymbol{R}^a - \gamma \boldsymbol{P}^a \boldsymbol{\Phi w}\}| \\
&= \max\big(\max_s \max_a \{\boldsymbol{R}^a + \gamma \boldsymbol{P}^a \boldsymbol{\Phi w} - \boldsymbol{\Phi w}\}\ , \\
&\qquad \max_s \min_a \{\boldsymbol{\Phi w} - \boldsymbol{R}^a - \gamma \boldsymbol{P}^a \boldsymbol{\Phi w}\}\big). \qquad (4.2)
\end{aligned}$$

The left hand side of the $\max$ operator in Equation 4.2 is efficiently computed under the stated assumptions by switching order of the $\max_s \max_a$ operators as $\max_a \max_s$. However, $\max_s \min_a$ cannot be switched with equality:

$$\max_s \min_a \{\boldsymbol{\Phi w} - \boldsymbol{R} - \gamma \boldsymbol{P}^a \boldsymbol{\Phi w}\} \leq \min_a \max_s \{\boldsymbol{\Phi w} - \boldsymbol{R} - \gamma \boldsymbol{P}^a \boldsymbol{\Phi w}\}, \qquad (4.3)$$

which results in the upper bound

$$\begin{aligned}
\text{BellErr}(\boldsymbol{\Phi w}) \leq\ & \max\big(\max_a \max_s \{\boldsymbol{R} + \gamma \boldsymbol{P}^a \boldsymbol{\Phi w} - \boldsymbol{\Phi w}\}\ , \\
& \min_a \max_s \{\boldsymbol{\Phi w} - \boldsymbol{R} - \gamma \boldsymbol{P}^a \boldsymbol{\Phi w}\}\big). \qquad (4.4)
\end{aligned}$$

Results and Discussion

The approximate policy iteration with compact constraints introduced by Guestrin *et al.* [27] is labeled API and this is the state of the art method we are comparing against. A variation of approximate policy iteration (which we thought of trying for the sake of completeness), using constraint generation instead of compact constraints, is labeled APIgen. The plain approximate linear programming of Algorithm 6 which builds and uses the entire constraint set is labeled ALP. The approximate linear program using constraint generation is labeled ALPgen.

So far we have not addressed the issue of choosing the set of basis functions. Guestrin *et al.* have used in their experiments the set of constant basis function ($\phi_0(\boldsymbol{x}) = 1, \forall \boldsymbol{x}$) plus basis indicator functions ($\phi_i(\boldsymbol{x}) = x_i, \forall \boldsymbol{x}, i = 1, \ldots, |\boldsymbol{x}|$). To facilitate a comparison with

their results, APIgen, ALP, and ALPgen all use the constant and indicator basis functions. However, as a prelude to the next chapter, to empirically verify the effect of expanding the set of basis functions we also report results obtained using approximate linear programming with constraints generation (ALPgen2) which, in addition to the constant and indicator basis functions, it also uses a set of basis functions over pairs of adjacent state variables. This set includes, for each pair of adjacent variables (x_i, x_{i+1}), the four basis functions (conjunctions):

$$\begin{aligned}
\phi_{k_1}(\boldsymbol{x}) &= x_i \wedge x_{i+1} \\
\phi_{k_2}(\boldsymbol{x}) &= \neg x_i \wedge x_{i+1} \\
\phi_{k_3}(\boldsymbol{x}) &= x_i \wedge \neg x_{i+1} \\
\phi_{k_4}(\boldsymbol{x}) &= \neg x_i \wedge \neg x_{i+1}.
\end{aligned}$$

The principal feature of the two problems we solve, also used in simulations by Guestrin *et al.* , is that, although the structure they exhibit fits very well the methods we want to compare, the problems are general enough and easily scalable to warrant drawing more than just narrow conclusions.

Although we experimented with domains of over 100 variables, we only report results for up to $n \leq 40$ variables so that we compare with the results of Guestrin *et al.* [27]. A number of quantities of interest are shown in Tables 4.1 and 4.2. In both problems, `cycle` and the `3legs` , the number of total constraints ultimately satisfied by the linear program is lowest by a significant margin for ALPgen. One can also observe that the technique of constraint generation is useful when used with approximate policy iteration in that it reduces the number of constraints its linear program would have to satisfy.

Although ALPgen iteratively expands the constraint set and resolves the LP it does show the lowest CPU times, with dramatic savings such as from 7.5 hours for API to 7 seconds for ALPgen. This performance increase does come at a price though.

Using the normalized upper bound as an error measure it can be seen that ALPgen has about twice the error of API.[2]

However, since ALPgen is much faster than API, the question arises whether one can

[1]These numbers are estimated from published results [27] (graphs). The exact probabilities and computer used for the simulations were not reported in that paper, so we cannot assert an exact comparison. However, perturbed probabilities have little effect on the performance of the methods we tried, and it seems that overall this is a loosely representative comparison of the general performance of the various algorithms on these problems.

[2]This is to be expected, since ALPgen minimizes an upper bound on the L_1 approximation error whereas API minimizes the Bellman error itself.

The cycle Problem—Time

$n =$	12	16	20	24	28	32	36	40
$N =$	4e3	6e4	1e6	2e7	3e8	4e9	7e10	1e12
API[1]	7m	30m	50m	1.3h	1.9h	3h	4.5h	7.5h
APIgen	39s	1.5m	2.3m	4.0m	6.5m	13m	22m	28m
ALP	4.5s	23s	1.4m	4.1m	10m	23m	47m	2.4h
ALPgen	0.7s	1.2s	1.8s	2.6s	3.5s	4.5s	5.9s	7.0s
ALPgen2	14s	37s	1.2m	2.8m	4.7m	6.4m	12m	17m

The cycle Problem—Number Constraints

$n =$	12	16	20	24	28	32	36	40
$N =$	4e3	6e4	1e6	2e7	3e8	4e9	7e10	1e12
APIgen	420	777	921	1270	1591	2747	4325	4438
ALP	1131	2023	3171	4575	6235	8151	10K	13K
ALPgen	38	50	62	74	86	98	110	122
ALPgen2	166	321	514	914	1223	1433	1951	2310

The cycle Problem—Bound on Bellman Error

$n =$	12	16	20	24	28	32	36	40
$N =$	4e3	6e4	1e6	2e7	3e8	4e9	7e10	1e12
API	0.36	0.34	0.33	0.33	0.32	0.32	0.32	0.31
APIgen	0.36	0.34	0.33	0.33	0.32	0.32	0.32	0.31
ALP	0.85	0.82	0.80	0.78	0.78	0.77	0.76	0.76
ALPgen	0.85	0.82	0.80	0.78	0.78	0.77	0.76	0.76
ALPgen2	0.12	0.14	0.08	0.08	0.10	0.08	0.07	0.07

Table 4.1: Table of results using ALP and constraint generation for the "system administrator" domain, in a cycle configuration.

The 3legs Problem—Time

$n =$	13	16	22	28	34	40
$N =$	8e4	6e4	4e6	3e8	2e10	1e12
API[1]	5m	15m	50m	1.3h	2.7h	5h
APIgen	28s	1.6m	3.9m	12m	23m	33m
ALP	0.7s	1.6s	6.0s	20s	56s	2.2m
ALPgen	0.7s	1.0s	1.5s	2.4s	3.4s	4.7s
ALPgen2	17s	33s	1.9m	5.4m	9.6m	23m

The 3legs Problem—Number Constraints

$n =$	13	16	22	28	34	40
$N =$	8e4	6e4	4e6	3e8	2e10	1e12
APIgen	363	952	1699	3792	6196	7636
ALP	729	1089	2025	3249	4761	6561
ALPgen	50	69	90	114	135	162
ALPgen2	261	381	826	1505	1925	3034

The 3legs Problem—Bound on Bellman Error

$n =$	13	16	22	28	34	40
$N =$	8e4	6e4	4e6	3e8	2e10	1e12
API	0.50	0.46	0.42	0.39	0.38	0.37
APIgen	0.50	0.46	0.42	0.39	0.38	0.37
ALP	0.96	0.82	0.78	0.78	0.77	0.76
ALPgen	0.96	0.82	0.78	0.78	0.77	0.76
ALPgen2	0.21	0.22	0.15	0.06	0.07	0.03

Table 4.2: Table of results using ALP and constraint generation for the "system administrator" domain, in a 3legs configuration.

improve the accuracy of the ALPgen's approximation by allowing it to use more basis functions, as implemented by ALPgen2. Indeed ALPgen2's solving time is longer than ALPgen, but still very competitive with the other methods. Its approximation error as measured by the upper bound is however brought to about half or better than that of API.

These experiments are by no means exhaustive, rather, while trying to investigate the practicality of using linear programming in conjunction with constraint generation to produce an approximate value function, they show a promising trend. In practice the algorithm seems to be computationally very competitive with existing algorithms while producing similar or better approximations. Furthermore, since solving the linear program does not seem to be the bottleneck, the approximation error can be reduced by expanding the set of basis functions and resolving the LP. However, this begs the question of where the additional basis functions should come from, or, similarly, what the provenance of the initial set of basis functions is. In our experiments we arbitrarily chose a set of basis functions to use, but how is one to know which basis functions to add to the current set so that the error decreases? The next chapter attempts to answer these and other related questions.

4.4 Obtaining an Approximation Using Branch-and-Bound Approximate Policy Iteration

Another important question, obtaining an approximate solution via minimizing the Bellman error, remains an open problem to the best of our knowledge, and we do not have an efficient exact method. Nevertheless, a similar branch-and-bound search strategy as used for calculating the Bellman error (described in Section 3.2) can be used to implement an approximate policy iteration scheme. The procedure is shown at a high level in Algorithm 9. It uses a linear program to recover the weight vector $\boldsymbol{w}^{(i+1)}$ that minimizes the one step error in approximating the value of the current policy $\pi_{\boldsymbol{w}^{(i)}}$ defined by $\boldsymbol{w}^{(i)}$. This linear program is indeed very large and would require enumerating constraints for all variable configurations (states), however, not all constraints are needed for the optimal solution. Therefore, if we could generate constraints efficiently, then we could obtain the optimal solution just as we did before. The other wrinkle is that we do not want to represent $\pi_{\boldsymbol{w}^{(i)}}$ explicitly, which makes it hard to see how this could be done if the constraints in Equation 4.5 remain written so. This part requires some further explanation.

Algorithm 9 Branch-and-Bound Approximate Policy Iteration (BB-API)

1: Start with an arbitrary $\boldsymbol{w}^{(0)}$, $i \leftarrow 0$.

2: **repeat**

3: Solve the linear program for $\boldsymbol{w}$ and δ:

$$\begin{aligned} \boldsymbol{w}^{(i+1)} \quad \leftarrow \quad & \underset{(\boldsymbol{w},\delta)}{\arg\min}\, \delta \quad \text{subject to:} \\ & \left.\begin{aligned} & v(\boldsymbol{w},\boldsymbol{x}) - q(\boldsymbol{w},\boldsymbol{x},\pi_{\boldsymbol{w}^{(i)}}(\boldsymbol{x})) \leq \delta \\ & q(\boldsymbol{w},\boldsymbol{x},\pi_{\boldsymbol{w}^{(i)}}(\boldsymbol{x})) - v(\boldsymbol{w},\boldsymbol{x}) \leq \delta \end{aligned}\right\} \text{ for all } \boldsymbol{x} \qquad (4.5) \\ & \delta \geq 0 \qquad (4.6) \end{aligned}$$

4: $i \leftarrow i + 1$.

5: **until** $\boldsymbol{w}^{(i+1)} = \boldsymbol{w}^{(i)}$

Here the process of constraint generation consists of two searches: one easy,

$$\max_{\boldsymbol{x}}\ q(\boldsymbol{w},\boldsymbol{x},\pi_{\boldsymbol{w}^{(i)}}(\boldsymbol{x})) - v(\boldsymbol{w},\boldsymbol{x}) - \delta,$$

and one hard,

$$\max_{\boldsymbol{x}}\ v(\boldsymbol{w},\boldsymbol{x}) - q(\boldsymbol{w},\boldsymbol{x},\pi_{\boldsymbol{w}^{(i)}}(\boldsymbol{x})) - \delta.$$

In both cases given are the current intermediate solution, $\boldsymbol{w}$, value function parameters from the previous step, $\boldsymbol{w}^{(i)}$, and the current intermediate solution δ.

The harder search is equivalent to

$$\max_{\boldsymbol{x}} \min_{a}\ v(\boldsymbol{w}^{(i)},\boldsymbol{x}) - q(\boldsymbol{w}^{(i)},\boldsymbol{x},a) - \delta,$$

and can be done using the branch-and-bound procedure already outlined in Section 3.2. Note the subtlety that, although the search uses parameters from the previous step—so as to find the action corresponding to $\pi_{\boldsymbol{w}^{(i)}}$, the constraint that is added to the current set will use the current LP solution $\boldsymbol{w}$. In essence, the closest we come to representing the greedy policy of the previous iteration explicitly is the set of constraints, one for each generated state $\boldsymbol{x}_g$ and its greedily best action with respect to the value function induced by $\boldsymbol{w}^{(i)}$, $\pi_{\boldsymbol{w}^{(i)}}(\boldsymbol{x}_g)$.

If one does not want to pay the price of a full branch-and-bound scheme, then a cheaper heuristic search for maximally violated constraints can be conducted by just running subgradient optimization until it stops making progress in reducing the upper bound. In fact,

this can lead to an effective way to calculate the initial assignment to $\boldsymbol{x}$ for a full branch-and-bound search: just start out with the best feasible solution $\boldsymbol{x}^t$ encountered during a preliminary subgradient optimization search. [3]

The overall procedure generalizes that of Guestrin *et al.* [27], and produces the same solutions in cases where both apply. However, this new technique does not require the additional assumption of a "default action" nor an explicit representation of the intermediate policies (in their case, a decision list). The drawback is that one has to perform a branch and bound search instead of solving cost networks to generate the constraints.

4.4.1 Experiments Using BB-API

We have tested the branch-and-bound approximate policy iteration procedure (BB-API) on two instances of the system administrator problem domain, namely the `cycle` and `3legs` problems with up to 10 state variables, and a discount factor of 0.95. Full description of the domain can be found in Appendix A.1. The experiments were encoded and run in the Matlab environment and carried out on 750MHz Pentium III level machines. We used the CPLEX 7.5 optimization suite to solve linear programs whenever it was needed.

The objective of these experiments was to obtain an error bound and run time trends for problem instances of increasing sizes. Unfortunately, more than ten state variables (n) tended to make the run times prohibitively long. Even so, we can easily see a trend developing in Table 4.3. For these experiments we used singleton identity basis functions; i.e. the constant function ($\phi_0(\boldsymbol{x}) = 1$) plus a function for each state variables ($\phi_i(\boldsymbol{x}) = x_i$), defined to return the variable's value. For each instance we report the exact Bellman error of the approximate solution as calculated by the branch-and-bound method, the time in seconds, and the number of policy iterations. In addition, we also report what we call the *practicality score* (Pract. Score) which tries to convey how practical the methods is, with higher score being better. This score is computed as the inverse of the product of error and time (in seconds).[4]

The results in Table 4.3 show that the BB-API is, as expected, much more expensive than straight approximate linear programming (ALP), but clearly produces better approximations given the same set of basis functions. Although BB-API's error is about half that of ALP,

[3] Sometimes a subgradient optimization search can accidentally solve the constrained optimization problem, and moreover know that it has the optimum.

[4] Assuming the exclusion of algorithms which take zero seconds to finish.

we need to keep in mind that the branch-and-bound method directly tries to minimize the Bellman error, whereas ALP actually minimizes the L_1 error. We have not compared the two methods in terms of their L_1 error, though it might give interesting results. To give an idea how large the worst error is, we mention that it comprised about 5% of the largest state value.

Another important fact to remember is that BB-API does not fully minimize the Bellman error of the final weight vector. Instead it achieves a bounded approximation of the Bellman error of the optimal weight vector [27]. Overall, as the exponential trend in run time seems to suggest, this method appears to be too costly in comparison with ALP to justify the modest gains in accuracy it offers. Another summary number that supports this conclusion is the average practicality score, which is higher for ALP than for BB-API (.15 versus .0022).

	ALP			BB-API			
n	B.Err.	Time (s)	Pract. Score	B.Err.	Time (s)	Pract. Score	Iters.
`cycle` problem, singleton bases							
5	2.8	2	0.18	0.9	160	0.00694	10
8	4.1	8	0.03	1.8	1,600	0.00034	16
10	6.7	14	0.01	2.4	5,672	0.00007	22
`3legs` problem, singleton bases							
4	1.8	1	0.55	0.6	383	.00435	7
7	4.0	3	0.08	1.0	697	0.00143	14
10	3.9	9	0.03	1.8	16,357	0.00003	19

Table 4.3: API versus ALP results

4.5 Obtaining an Approximation Using the Dual LP

Another way of obtaining an approximate solution, a trivial derivative of previously mentioned methods, has remained unexplored in the literature. This method involves the use of the dual linear program. Although we do not promote this method, we feel that our attempt should be mentioned as a potential warning sign regarding its impracticality. As far as we know an empirical investigation of the direct approximation based on the dual LP has not

been published.[5]

Recall that in Algorithm 3 we presented the linear programming formulation of solving an MDP. The dual method can be derived from either probability principles, or from duality theory. We will derive these next.

4.5.1 Deriving the Dual

We adapt here a formulation from [52]. Suppose the initial states of the MDP are drawn from the probability distribution ν, that is,

$$P(s_0) = \nu(s_0)$$

We want to find the policy π which maximizes

$$E\left\{\sum_{n=0}^{\infty} \gamma^n R(s_n, \pi(s_n))\right\}$$

In this derivation we will allow for stochastic policies. That is, the action a in state s is drawn from a distribution over actions $P(a|s) = \pi(a|s)$. For a given policy π let y_{sa} denote the expected discounted time that the process is in state s and action a is chosen. That is,

$$y_{sa} = E_\pi\left\{\sum_{n=0}^{\infty} \gamma^n I_{\{s_n=s, a_n=a\}}\right\} \tag{4.7}$$

where for any event B,

$$I_B = \begin{cases} 1, & \text{if } B \text{ occurs} \\ 0, & \text{otherwise} \end{cases}$$

First, we want to derive the time that the process spends in a given state s, when acting according to policy π:

[5] Guestrin [26] has investigated the approximate dual LP theoretically.

$$
\begin{aligned}
\sum_a y_{s'a} &= \sum_a E_\pi \left\{ \sum_{n=0}^{\infty} \gamma^n I_{\{s_n=s', a_n=a\}} \right\} \\
&= \sum_a E_\pi \left\{ \gamma^0 I_{\{s_0=s', a_0=a\}} + \gamma^1 I_{\{s_1=s', a_1=a\}} + \gamma^2 I_{\{s_2=s', a_2=a\}} + \cdots \right\} \\
&= \sum_a (\gamma^0 E_\pi[I_{\{s_0=s', a_0=a\}}] + \gamma^1 E_\pi[I_{\{s_1=s', a_1=a\}}] + \\
&\quad \gamma^2 E_\pi[I_{\{s_2=s', a_2=a\}}] + \cdots) \\
&= \sum_a (\gamma^0 P(s_0 = s', a_0 = a) + \gamma^1 P(s_1 = s', a_1 = a) + \\
&\quad \gamma^2 P(s_2 = s', a_2 = a) + \cdots) \qquad (4.8) \\
&= \gamma^0 P(s_0 = s') + \gamma^1 P(s_1 = s') + \gamma^2 P(s_2 = s') + \cdots) \qquad (4.9) \\
&= \gamma^0 E[I_{\{s_0=s'\}}] + \gamma^1 E[I_{\{s_1=s'\}}] + \gamma^2 E[I_{\{s_2=s'\}}] + \cdots \qquad (4.10) \\
&= E \left\{ \sum_{n=0}^{\infty} \gamma^n I_{\{s_n=s'\}} \right\}
\end{aligned}
$$

A few remarks:

- in Step 4.8 we use the definition the expected value with respect to the distribution induced by the policy and transitions;
- Step 4.9 marginalizes over the action variable,
- and Step 4.10 re-uses the expected value definition.

Now we will show what happens if we sum y_{sa} over both states and actions. Let us start from Step 4.9 above.

$$
\begin{aligned}
\sum_{s'} \sum_a y_{s'a} &= \sum_{s'} (\gamma^0 P(s_0 = s') + \gamma^1 P(s_1 = s') + \gamma^2 P(s_2 = s') + \cdots) \\
&= \gamma^0 + \gamma^1 + \gamma^2 + \cdots \\
&= \frac{1}{1-\gamma} \qquad (4.11)
\end{aligned}
$$

There are two more parts to the derivation of a linear programming method; to show that

$$
\sum_a y_{s'a} = \nu_{s'} + \gamma \sum_s \sum_a P(s'|s,a) y_{sa} \qquad (4.12)
$$

and that the numbers y_{sa} which satisfy Equations 4.11 and 4.12 can be interpreted as the expected discounted time that the process is in state s' and action a is chosen when the initial state is chosen according to the probabilities ν and the policy π given by

$$\pi(s,a) = \frac{y_{sa}}{\sum_{a'} y_{sa'}} \tag{4.13}$$

is used. To derive the first part we start from Step 4.10:

$$\begin{aligned}
\sum_a y_{s'a} &= \gamma^0 E[I_{\{s_0=s'\}}] + \gamma^1 E[I_{\{s_1=s'\}}] + \gamma^2 E[I_{\{s_2=s'\}}] + \cdots && (4.14)\\
&= \gamma^0 P(s_0 = s') + \\
&\quad \gamma^1 \sum_s \sum_a E[I_{\{s_0=s,a_0=a\}}]\, P(s_1 = s' | s_0 = s, a_0 = a) + \\
&\quad \gamma^2 \sum_s \sum_a E[I_{\{s_2=s',a_2=a\}}]\, P(s_2 = s' | s_1 = s, a_1 = a) + \cdots && (4.15)\\
&= \gamma^0 P(s_0 = s') + \gamma \sum_s \sum_a P(s'|s,a) \sum_{n=0}^{\infty} E[\gamma^n I_{\{s_n=s,a_n=a\}}] && (4.16)\\
&= \gamma^0 P(s_0 = s') + \gamma \sum_s \sum_a P(s'|s,a) E[\sum_{n=0}^{\infty} \gamma^n I_{\{s_n=s,a_n=a\}}] && (4.17)\\
&= \nu(s') + \gamma \sum_s \sum_a P(s'|s,a) y_{sa} && (4.18)
\end{aligned}$$

Going from Step 4.14 to 4.15 is based on the following identity:

$$I_{\{s_{n+1}=s'\}} = \sum_s \sum_a I_{\{s_n=s,a_n=a\}} I_{\{s_{n+1}=s'\}}$$

Taking expectations of both sides, we obtain:

$$\begin{aligned}
E[I_{\{s_{n+1}=s'\}}] &= \sum_s \sum_a E[I_{\{s_n=s,a_n=a\}} I_{\{s_{n+1}=s'\}}] \\
&= \sum_s \sum_a P(s_{n+1} = s', s_n = s, a_n = a) \\
&= \sum_s \sum_a P(s_{n+1} = s' | s_n = s, a_n = a) P(s_n = s, a_n = a) \\
&= \sum_s \sum_a P(s_{n+1} = s' | s_n = s, a_n = a) E[I_{\{s_n=s,a_n=a\}}]
\end{aligned} \tag{4.19}$$

Then, for Step 4.16 we observe that the transition probabilities are not time dependent, as the process is stationary, and $P(s'|s,a)$ can be factored out of the infinite sum. The last step, 4.18 uses the definition of y_{sa} given in 4.7.

Therefore the set of numbers y_{sa}, i.e. one number for each (s,a) pair, which satisfy Equations 4.11 and 4.12 can be interpreted as the expected discounted time that the process is in state s' and action a is chosen when the initial state is chosen with probability $\nu_{s'}$ and the policy π is given by:

$$\pi(s,a) = \frac{y_{sa}}{\sum_{a'} y_{sa'}}.$$

However, we need to find the set of numbers y_{sa} corresponding to the optimal policy, and given the definition of $y_{sa'}$, we need to solve the following linear program:

$$\begin{aligned}
\max_{y_{s'a}, a\in A, s'\in S} \quad & \sum_a \sum_{s'} y_{s'a} r(s',a) \\
\text{subject to:} \quad & \sum_a \sum_{s'} y_{s'a} = \frac{1}{1-\gamma} \\
& y_{s'a} \geq 0, \quad \forall s', a \\
& \sum_a y_{s'a} = \nu(s') + \gamma \sum_s \sum_a P(s'|s,a) y_{sa}, \quad \forall s'
\end{aligned} \tag{4.20}$$

Before we do that, we need to draw a connection between the primal and dual LPs. First we will re-formulate 4.20 in matrix form. Let $\boldsymbol{y}$ be a vector of NM variables, $\boldsymbol{y} = \begin{bmatrix} \boldsymbol{y}^{a_1} \\ \boldsymbol{y}^{a_2} \\ \vdots \\ \boldsymbol{y}^{a_M} \end{bmatrix}$ where each $\boldsymbol{y}^{a_j}$ is a $N \times 1$ vector. Let

$$C = \begin{bmatrix} I \\ I \\ \vdots \\ I \end{bmatrix} - \gamma \begin{bmatrix} \boldsymbol{P}^{a_1} \\ \boldsymbol{P}^{a_2} \\ \vdots \\ \boldsymbol{P}^{a_M} \end{bmatrix}, \text{ and } \boldsymbol{r} = \begin{bmatrix} \boldsymbol{R}^{a_1} \\ \boldsymbol{R}^{a_2} \\ \vdots \\ \boldsymbol{R}^{a_M} \end{bmatrix}$$

where each $\boldsymbol{R}^{a_j}$ is a $N \times 1$ vector of rewards.

Then 4.20 can be written as:

$$\begin{aligned} \max_{\boldsymbol{y}} \quad & \boldsymbol{r}^T \boldsymbol{y} \\ \text{subject to:} \quad & C^T \boldsymbol{y} = \boldsymbol{\nu} \\ & \boldsymbol{y} \geq \mathbf{0} \end{aligned} \tag{4.21}$$

For the moment we will leave the constraint $\sum_a \sum_{s'} y_{s'a} = \frac{1}{1-\gamma}$ out.

Suppose we want an upper bound on the objective of 4.21, $\boldsymbol{r}^T \boldsymbol{y}$. Then, for any $N \times 1$ vector $\boldsymbol{v}$, the quantity

$$\boldsymbol{r}^T \boldsymbol{y} + \boldsymbol{v}^T (\boldsymbol{\nu} - C^T \boldsymbol{y}) \tag{4.22}$$

is an upper bound. Re-arranging Equation 4.22 yields

$$\boldsymbol{v}^T \boldsymbol{\nu} + (\boldsymbol{r}^T - \boldsymbol{v}^T C^T) \boldsymbol{y} \tag{4.23}$$

However, this upper bound can be driven to $+\infty$ unless we restrict $\boldsymbol{v}$ such that $(\boldsymbol{r}^T - \boldsymbol{v}^T C^T) \leq \mathbf{0}$ and thus $C\boldsymbol{v} \geq \boldsymbol{r}$. Furthermore, we are interested in an instance of $\boldsymbol{v}$ which gives us the *smallest* upper bound. To achieve this we either minimize 4.23 or just $\boldsymbol{v}^T \boldsymbol{\nu}$ which remains an upper bound if $C\boldsymbol{v} \geq \boldsymbol{r}$. Putting all together, we now have a linear programming optimization which is almost identical to the primal LP we used earlier. The only difference is the use of the initial probability distribution $\boldsymbol{\nu}$ in the LP objective.

$$\begin{aligned} \min_{\boldsymbol{v}} \quad & \boldsymbol{\nu}^T \boldsymbol{v} \\ \text{subject to:} \quad & C\boldsymbol{v} \leq \boldsymbol{r} \end{aligned} \tag{4.24}$$

In summary, we derived the dual LP from first principles. Then, while minimizing an upper bound on the dual objective, we arrived at the primal LP. Note that we left out one of the constraints, which does not change the resulting optimal policy solution. This holds true also whether we use $\boldsymbol{\nu}$ or $\mathbf{1}$ as coefficients in the primal objective, because there are no artificially imposed restrictions on the solution. If we restricted the value function to some particular class of functions, such as that spanned by a set of basis functions, then the choice of coefficients in the primal objective can make a difference [15].

From linear programming fundamentals we know that the primal can be solved and the dual solution is just the primal Lagrange multipliers (sometimes referred to as prices), or vice versa, the dual can be solved and the primal solution is the dual Lagrange multipliers.

Given the primal LP, the dual LP can be easily derived. From now on, unless otherwise stated, the primal LP will be just as we have used it in earlier chapters,

$$\begin{aligned} \min_{\boldsymbol{v}} \quad & \mathbf{1}^T \boldsymbol{v} \\ \text{subject to:} \quad & C\boldsymbol{v} \leq \boldsymbol{r} \end{aligned} \tag{4.25}$$

and its dual will be:

$$\begin{aligned} \max_{\boldsymbol{y}} \quad & \boldsymbol{r}^T \boldsymbol{y} \\ \text{subject to:} \quad & C^T \boldsymbol{y} = \mathbf{1} \\ & \boldsymbol{y} \geq \mathbf{0} \end{aligned} \tag{4.26}$$

4.5.2 Approximating the Dual

We are ready now to develop an approximate method for solving MDPs. We start with the dual LP from 4.26 and replace variables $\boldsymbol{y}$ by a weighted combination of basis functions, such that $y_{sa} \approx \sum_k \phi_{ka}(s) u_{ka}$. Thus each action will have its set of basis functions and its own variables. The approximate dual LP will then have only KM variables, instead of NM.[6] Tentatively, we could just replace the exact $\boldsymbol{y}$ variables with the approximate version and leave the exact dual LP as is:

$$\begin{aligned} \max_{\boldsymbol{u}} \quad & \boldsymbol{r}^T \boldsymbol{\Upsilon} \boldsymbol{u} \\ \text{subject to:} \quad & C^T \boldsymbol{\Upsilon} \boldsymbol{u} = \mathbf{1} \\ & \boldsymbol{\Upsilon} \boldsymbol{u} \geq \mathbf{0} \end{aligned} \tag{4.27}$$

where

$$\boldsymbol{\Upsilon} = [\Phi^{a_1} \ \cdots \ \Phi^{a_M}]$$

and $\boldsymbol{u} = \begin{bmatrix} \boldsymbol{u}^{a_1} \\ \boldsymbol{u}^{a_2} \\ \vdots \\ \boldsymbol{u}^{a_M} \end{bmatrix}$ such that each $\boldsymbol{u}^{a_j}$ is a $K \times 1$ vector.

[6] This is true if, for simplicity, we assume K basis functions per action. However, we could instead have a different number of basis functions for each action, and then we would need a variable for each action and basis function.

The difficulty is clear immediately: the strict equality constraints might make the LP infeasible if the set of basis functions is not accommodating enough. Instead, we could just minimize the amount by which constraints are not satisfied:

$$\begin{aligned} \max_{\boldsymbol{u},\delta} \quad & \boldsymbol{r}^T \boldsymbol{\Upsilon} \boldsymbol{u} - \delta \\ \text{subject to:} \quad & |C^T \boldsymbol{\Upsilon} \boldsymbol{u} - \mathbf{1}| \leq \delta \\ & \delta \geq 0 \\ & \boldsymbol{\Upsilon} \boldsymbol{u} \geq \mathbf{0} \end{aligned} \tag{4.28}$$

or

$$\begin{aligned} \max_{\boldsymbol{u},\delta} \quad & \boldsymbol{r}^T \boldsymbol{\Upsilon} \boldsymbol{u} - \delta \\ \text{subject to:} \quad & C^T \boldsymbol{\Upsilon} \boldsymbol{u} - \mathbf{1} \leq \delta \\ & \mathbf{1} - C^T \boldsymbol{\Upsilon} \boldsymbol{u} \leq \delta \\ & \delta \geq 0 \\ & \boldsymbol{\Upsilon} \boldsymbol{u} \geq \mathbf{0} \end{aligned} \tag{4.29}$$

Recall that the policy induced by a dual solution $\boldsymbol{y}$ is defined as:

$$\pi(s,a) = \frac{y_{sa}}{\sum_{a'} y_{sa'}}$$

Similarly, given a solution $\boldsymbol{u}$ for the approximate dual (4.29), the induced randomized policy would be:

$$\pi_{\boldsymbol{u}}(s,a) = \frac{\sum_k \phi_{ka}(s) u_{ka}}{\sum_{a'} \sum_k \phi_{ka'}(s) u_{ka'}}$$

4.5.3 Experiments with the Dual LP Approximation

Although we ran simulations on the `cycle`, `3legs`, and `3loops` problems, we omit showing here the results; rather, we will only comment on them. The dual LP approximating method was competitive with the primal based methods in terms of run times, but produced approximate solutions with much larger error. We do not fully understand why this would be the case, but, from preliminary experimentation, we deduced that the exact dual LP solution, which can be interpreted as a discounted state visitation count, is much harder to represent by a linear combination of basis functions than the exact solution of the primal

LP (i.e. the value function). This became clear once we tried to fit the exact dual solution using regression. Unfortunately, it seems that the exact dual solution lives in a space far from that spanned by the basis functions we tried. Given these negative results, we feel that it is fair to mention them here as a warning that further exploration of this direction may not be warranted.

Chapter 5

Improving the Approximate Solution

5.1 Introduction

While approximately solving an MDP from scratch is desirable, we would also like to be able to improve an existing solution, regardless of its origin. There are several reasons for wanting to improve a given solution; for instance it might come from an unreliable source, in which case we may not want to assume that its error is small, or perhaps we made the effort and calculated its large error using the branch-and-bound technique of Section 3.2.

Again, we adopt the regular assumptions of previous sections, that the MDP is compactly described using a DBN and the solution is based on a bunch of basis functions. Thus, given the basis functions and assuming that the MDP has been solved for an approximate solution already, the question is: how to obtain a new solution with a smaller error. Another debate can ensue on the relative merit of different error measures, and which one should be decreased by the new solution. Although it is desirable that a new solution would bring a decrease in all error measurements, this would be an unrealistic expectation. While—based on its relation to the L_∞ error—some prefer that the Bellman error is decreased, we have already seen how computationally inefficient this can be, even with improved algorithms such as the branch-and-bound technique of Section 3.2. Instead, encouraged by the success of minimizing the weighted L_1 error in the previous chapter, we opt in its favor for the new task of improving an existing solution. Note that no ideals have been given up with certainty. In fact, significantly reducing the L_1 error is bound to eventually reduce the other errors as well.[1]

The question then is: *how* to reduce the error? Since the approximate linear programming method coupled with constraint generation showed promising results, we start with it as the basis for a new algorithm. Recall that as soon as the approximate LP has been solved, there is nothing else to be done. That is, the optimization has a clearly defined stopping criterion: when no more constraints can be generated (are violated). There are two directions that can be explored from here:

1. find a new vector of coefficients for the LP objective (*state relevance weights*), or

2. find new basis functions to augment the current set.

De Farias and van Roy [15] show that the choice of state relevance weights can be impor-

[1] Intuitively, if the L_1 error is reduced below the maximum per state error (L_∞) then all other errors must have been reduced.

tant, and point to heuristics in choosing these weights, though, how practical and efficacious these heuristics are is not clear. Besides common sense being in favor of searching for more basis functions, we also produced evidence in the last chapter that more basis functions can drastically reduce some error bound. Another advantage of improving the solution with new basis functions is that, in theory, one can keep on adding basis functions until the error drops below a threshold or vanishes. Although impractical, this point is based on the fact that a finite number of basis functions is sufficient for an exact solution.

We are not aware of any procedure that can find the optimal state relevance weights. In this chapter we present methods for discovering and producing basis functions with the goal of reducing the approximation error.

The rest of the chapter is comprised of Section 5.2, which introduces the last contribution of this thesis, namely the algorithm to produce new basis functions such that the approximation error is reduced.

5.2 Incrementing the Set of Basis Functions

In this section we develop procedures for creating and comparing new basis functions. Suppose we are given a basis function. We do not know what impact it will have on the solution should we decide to use it, unless we re-solve the LP and calculate the error of the new solution. However, since the concrete goal of the optimization is to minimize the LP objective, why not use it as a less expensive means of gauging the goodness of a basis function, instead of a full blown branch-and-bound error calculation? Thus, we want to find new basis functions which, when used in the value function representation, improve (i.e. reduce) the linear program objective. Whether one can improve it optimally or not is not clear yet.

Recall that a basis function is defined over a set of multivalued variables whose totality of configurations comprises the domain of the function, and returns a scalar for each of these configurations. Therefore, whenever we talk about creating a basis function, we mean that we produce a subset of state variables,[2] and a set of values, one for each configurations of variables. As mentioned before, we also want to be able to order basis functions, i.e. to associate a value with each, which would be indicative of how desirable the basis function is.

[2]We abuse the meaning of domain by sometimes referring to the set of variables as the domain, instead of explicitly saying the set of configurations of variables.

In summary, there are three issues when considering the creation (or discovery) and addition of a basis function:

- decide state variables that should be part of its domain,
- construct a basis function given the candidate domain, i.e. produce a table of real values, one per configuration,
- and score the effect of using the candidate basis function.

Putting it all together leads effortlessly to a high level method as shown in Algorithm 10. Perhaps some steps in this algorithm can be collapsed into one (e.g. generating the basis

Algorithm 10 Expanding The Basis Function Set

$\boldsymbol{B} \leftarrow \boldsymbol{\phi}_0, w^{(0)} \leftarrow \mathbf{0}, i \leftarrow 0$
while NOT SATISFIED WITH SOLUTION $\boldsymbol{w}^{(i)}$ **do**
  $i \leftarrow i + 1$
  generate candidate domains $\boldsymbol{x}_j \subseteq \boldsymbol{x}$, where $j = 1, 2, \ldots, J$
  for each $\boldsymbol{x}_j, j = 1, 2, \ldots, J$ **do**
    construct basis function $\boldsymbol{\phi}_{i_j}(\boldsymbol{x}_j)$
    score ability of $\boldsymbol{\phi}_{i_j}(\boldsymbol{x}_j)$ to reduce LP objective
  $\boldsymbol{B} \leftarrow \boldsymbol{B} \cup \{\boldsymbol{\phi}_{i_k} : \text{score}(\boldsymbol{\phi}_{i_k}) \geq \text{score}(\boldsymbol{\phi}_{i_j}), j = 1, \ldots, k-1, k+1, \ldots, J.$
  $\boldsymbol{w}^{(i)} \leftarrow$ result of solving LP using the new set of basis functions $\boldsymbol{B}$

function values as well as producing a score), but that does not change the essence of the method.

The three main issues require individual attention.

5.2.1 Choosing a Basis Function Candidate Domain

In choosing candidate domains we want to maintain two requirements: candidate domain sizes should remain small, such that the cost networks do not become unwieldy, and the set of candidate domains should be exhaustive; i.e., eventually the procedure should produce a set that includes every choice of subset of state variables less than or equal to a given size—once only for binary variables.

With these goals in mind we propose three approaches that differ mainly in how aggressively they move from small domains to considering larger domains. The more aggressive the expansion, the more computationally expensive it is.

The first method, which we call *sequential*, is the most conservative; it first considers singleton domains until they are exhausted, then doubleton, and so on. The second approach, which we call *lattice* because its expansion follows a *subset of* lattice, considers a candidate domain if and only if all of its proper subsets have already been used as domains in the current set of basis functions. For instance, if the current set of domains was

$$\{\}, \{a\}, \{b\}, \{c\}, \{a, b\}, \{a, c\}$$

then $\{a, b, c\}$ could not be added to the set, but $\{b, c\}$ could. Only after adding a basis function defined on variables $\{b, c\}$, could another be added on $\{a, b, c\}$.

The least conservative approach, *neighbors*, considers all candidate domains which can be constructed as the union of one of its current domains and a new state variable.

5.2.2 Scoring the Basis Function

Although Algorithm 10 needs to construct a basis function before scoring it, it is essential to understand the scoring methods first.

Let us assume that the current set of basis functions is $\boldsymbol{\Phi} = [\boldsymbol{\phi}_1 \boldsymbol{\phi}_2 \cdots \boldsymbol{\phi}_{k-1}]$ and, as noted before, to obtain an approximate value function we must solve the linear program:

$$\min_{\boldsymbol{w}} \boldsymbol{\varphi}^T \boldsymbol{w} \text{ subject to } \boldsymbol{\Phi}\boldsymbol{w} \geq \boldsymbol{R}^a + \gamma \boldsymbol{P}^a \boldsymbol{\Phi}\boldsymbol{w} \quad \forall a$$

or, re-arranging,

$$\min_{\boldsymbol{w}} \boldsymbol{\varphi}^T \boldsymbol{w} \text{ subject to } \left(I - \gamma \boldsymbol{P}^a\right) \boldsymbol{\Phi}\boldsymbol{w} \geq \boldsymbol{R}^a \quad \forall a.$$

To further simplify notation let

$$\boldsymbol{C} = \begin{bmatrix} \left(I - \gamma \boldsymbol{P}^{a_1}\right) \boldsymbol{\Phi} \\ \left(I - \gamma \boldsymbol{P}^{a_2}\right) \boldsymbol{\Phi} \\ \vdots \\ \left(I - \gamma \boldsymbol{P}^{a_m}\right) \boldsymbol{\Phi} \end{bmatrix}$$

and

$$\boldsymbol{r} = \begin{bmatrix} \boldsymbol{R}^{a_1} \\ \boldsymbol{R}^{a_2} \\ \vdots \\ \boldsymbol{R}^{a_m} \end{bmatrix}$$

so that the LP becomes

$$\min_{\boldsymbol{w}} \boldsymbol{\varphi}^T \boldsymbol{w} \text{ subject to } \boldsymbol{C}\boldsymbol{w} \geq \boldsymbol{r}. \tag{5.1}$$

We will call $\boldsymbol{C}$ the coefficient constraint matrix, or, simply the constraint matrix. We will also call Equation 5.1 the *primal* of the linear program. From duality theory [12] we have that the *dual* of this LP is

$$\max_{\boldsymbol{\lambda}} \boldsymbol{\lambda}^T \boldsymbol{r} \text{ subject to } \boldsymbol{\lambda}^T \boldsymbol{C} = \boldsymbol{\varphi}, \boldsymbol{\lambda} \geq \boldsymbol{0}. \tag{5.2}$$

Assume we have solved the primal and dual linear program and obtained a solution pair $(\boldsymbol{w}, \boldsymbol{\lambda})$.

Now we want to measure the impact that adding a basis function $\boldsymbol{\phi}_k$ has on the linear program objective with current solution pair $(\boldsymbol{w}, \boldsymbol{\lambda})$ and the set $[\boldsymbol{\phi}_1 \boldsymbol{\phi}_2 \cdots \boldsymbol{\phi}_{k-1}]$ as current basis functions. The new basis function induces a new column $\boldsymbol{c}_{:,k}$ in the constraint matrix, and a new entry φ_k in the LP objective. A new LP variable w'_k is also needed. The new primal will be

$$\min_{\boldsymbol{w}'} \boldsymbol{\varphi}'^T \boldsymbol{w}' \text{ subject to } \boldsymbol{C}'\boldsymbol{w} \geq \boldsymbol{r}, \boldsymbol{w}' \geq \boldsymbol{0} \tag{5.3}$$

where

$$\begin{aligned} \boldsymbol{w}' &= [w'_1\, w'_2\, \cdots\, w'_k]^T \\ \boldsymbol{\varphi}' &= [\varphi_1\, \varphi_2\, \cdots\, \varphi_k]^T \\ \boldsymbol{C}' &= [\boldsymbol{c}_{:,1}\, \boldsymbol{c}_{:,2}\, \cdots \boldsymbol{c}_{:,k}]. \end{aligned}$$

The ideal scenario would be to add a basis function which reduces the LP objective maximally. That is, given the current LP solution $\boldsymbol{w}$ we would like a basis function $\boldsymbol{\phi}_k$ such that

$$\boldsymbol{\varphi}^T \boldsymbol{w} - \boldsymbol{\varphi}'^T \boldsymbol{w}' \geq \boldsymbol{\varphi}^T \boldsymbol{w} - \boldsymbol{\varphi}'^T \boldsymbol{u}',\ \forall \boldsymbol{u}'.$$

The obvious approach to achieve this goal is to simply add the new basis function to the current set and resolve the entire LP. Since our objective is to quantify this effect so that we

can compare candidate functions, we define score_flp($\boldsymbol{\phi}_k$) to be the value of the LP objective after solving it with the new basis function included:

$$\text{score\_flp}(\boldsymbol{\phi}_k) = \boldsymbol{\varphi}'^T \boldsymbol{w}^{*\prime}.$$

It should be clear that the main limitation of this approach is its cost. Resolving the entire linear program for each candidate function is hardly a desirable feature.

At the other end of the spectrum—as far as computation cost goes—stands an approach inspired from duality theory. This approach gives up the optimal reduction of the LP objective in favor of a potential reduction, no matter how small, but which is scored in a computationally efficient manner. Recall that, for a solution pair $(\boldsymbol{w}, \boldsymbol{\lambda})$, all primal constraints are satisfied. The column $\boldsymbol{c}_{:,k}$ induced by the newly added basis function $\boldsymbol{\phi}_k$ corresponds to the dual constraint $\boldsymbol{c}_{:,k}^T \boldsymbol{\lambda} = \varphi_k$. If this constraint is satisfied, then $\boldsymbol{\phi}_k$ is guaranteed not to provide any improvement in the LP solution. Therefore we define

$$\text{score\_dual}(\boldsymbol{\phi}_k) = \max \left\{ |\min_{\boldsymbol{c}_{:,k}, \varphi_k} \boldsymbol{c}_{:,k}^T \boldsymbol{\lambda} - \varphi_k|, |\max_{\boldsymbol{c}_{:,k}, \varphi_k} \boldsymbol{c}_{:,k}^T \boldsymbol{\lambda} - \varphi_k| \right\}.$$

Thus this score is higher if the new dual constraint is unsatisfied by a larger amount, on either side of 0. Note that whereas before $\boldsymbol{\lambda}$ were the unknowns, now $\boldsymbol{\lambda}$ are given, while $\boldsymbol{c}_{:,k}$ and φ_k are (indirectly) the variables. An implementation detail is that the real variables are, of course, the basis function parameters, or table values, which induce $\boldsymbol{c}_{:,k}$ and φ_k. The linear optimization needed to produce this score requires further treatment and is explained in detail in the next section, where it fits better conceptually.

For now, we propose yet another approach which offers a compromise—both in terms of computational cost and accuracy of prediction—between score_flp() and score_dual(). Assume again having a current LP solution pair $(\boldsymbol{w}, \boldsymbol{\lambda})$ and a candidate basis function $\boldsymbol{\phi}_k$.

According to complementary slackness properties from linear programming we have that:

$$\begin{aligned} &\text{if} \quad \lambda_i > 0 \quad \text{then} \quad \boldsymbol{c}_{i,:} \boldsymbol{w} = r_i \\ &\text{if} \quad \lambda_i = 0 \quad \text{then} \quad \boldsymbol{c}_{i,:} \boldsymbol{w} \geq r_i \end{aligned}$$

where $\boldsymbol{c}_{i,:}$ and $\boldsymbol{c}_{:,j}$ are the i-th row and j-th column of $\boldsymbol{C}$, and $i = 1, 2, \ldots, mN$, $j = 0, 1, \ldots, K$. In other words, the primal/dual constraints corresponding to non-zero solution parameters are satisfied with no slack (i.e. we have an equality expression) and are called active constraints. Let $B = \{i : \lambda_i > 0\}$ be the set of indexes of the primal active constraints,

then

$$\begin{aligned} \boldsymbol{C}_B \boldsymbol{w} &= \boldsymbol{r}_B \\ \boldsymbol{w} &= \boldsymbol{C}_B^{-1} \boldsymbol{r}_B \end{aligned} \tag{5.4}$$

where $\boldsymbol{C}_B$ is square and invertible for non-ill posed problems.

If we impose that the solution of the new LP—the one using $\boldsymbol{\phi}_k$ as well—has an active constraint set B' inclusive of the original constraint set B ($B \subset B'$), we then have the following linear program:

$$\min_{\boldsymbol{w'}} \sum_{j=0}^{k-1} \varphi_j w'_j + \varphi_k w'_k \text{ subject to } \begin{cases} \boldsymbol{C}'_B \boldsymbol{w'} = \boldsymbol{r}_B \\ \boldsymbol{C}'_N \boldsymbol{w'} \geq \boldsymbol{r}_N \\ \boldsymbol{w'} \geq \boldsymbol{0} \end{cases} \tag{5.5}$$

where $N = \{i : \lambda_i = 0\}$. The first constraint expression of this linear program can also be written as:

$$\begin{aligned} \boldsymbol{C}'_B \boldsymbol{w'} &= \boldsymbol{r}_B \\ \boldsymbol{C}_B \begin{bmatrix} w'_1 \\ w'_2 \\ \vdots \\ w'_{k-1} \end{bmatrix} + \boldsymbol{c}_{B,k} w'_k &= \boldsymbol{r}_B \\ \boldsymbol{C}_B \boldsymbol{w'}_{1:k-1} + \boldsymbol{c}_{B,k} w'_k &= \boldsymbol{r}_B \end{aligned}$$

from which we solve for $\boldsymbol{w'}_{1:k-1}$

$$\begin{aligned} \boldsymbol{w'}_{1:k-1} &= \boldsymbol{C}_B^{-1} \boldsymbol{r}_B - \boldsymbol{C}_B^{-1} \boldsymbol{c}_{B,k} w'_k \\ &= \boldsymbol{w} - \boldsymbol{C}_B^{-1} \boldsymbol{c}_{B,k} w'_k, \end{aligned} \tag{5.6}$$

where the last line follows from Equation 5.4. Re-writing Equation 5.5 as

$$\min_{\boldsymbol{w'}} \boldsymbol{\varphi}_{1:k-1}^T \boldsymbol{w'}_{1:k-1} + \varphi_k w'_k \text{ subject to } \begin{cases} \boldsymbol{C}_B \boldsymbol{w'}_{1:k-1} + \boldsymbol{c}_{B,k} w'_k = \boldsymbol{r}_B \\ \boldsymbol{C}_N \boldsymbol{w'}_{1:k-1} + \boldsymbol{c}_{N,k} w'_k \geq \boldsymbol{r}_N \\ \boldsymbol{w'} \geq \boldsymbol{0} \end{cases}$$

and substituting $\boldsymbol{w}'_{1:k-1}$ reduces the linear program from k variables to just one:

$$\begin{aligned}
&\min_{w'_k} \boldsymbol{\varphi}_{1:k-1}^T \left(\boldsymbol{w} - \boldsymbol{C}_B^{-1}\boldsymbol{c}_{B,k}w'_k\right) + \varphi_k w'_k \\
&\text{subject to} \\
&\begin{cases} \boldsymbol{C}_B \left(\boldsymbol{w} - \boldsymbol{C}_B^{-1}\boldsymbol{c}_{B,k}w'_k\right) + \boldsymbol{c}_{B,k}w'_k = \boldsymbol{r}_B \\ \boldsymbol{C}_N \left(\boldsymbol{w} - \boldsymbol{C}_B^{-1}\boldsymbol{c}_{B,k}w'_k\right) + \boldsymbol{c}_{N,k}w'_k \geq \boldsymbol{r}_N \\ \boldsymbol{w}' \geq \boldsymbol{0} \end{cases} .
\end{aligned} \tag{5.7}$$

This can be further simplified to finally obtain:

$$\begin{aligned}
&\min_{w'_k} \left(\varphi_k - \boldsymbol{\varphi}_{1:k-1}^T \boldsymbol{C}_B^{-1}\boldsymbol{c}_{B,k}w'_k\right) w'_k \\
&\text{subject to} \\
&\begin{cases} \left(\boldsymbol{c}_{N,k} - \boldsymbol{C}_N\boldsymbol{C}_B^{-1}\boldsymbol{c}_{B,k}\right) w'_k \geq \boldsymbol{r}_N - \boldsymbol{C}_N\boldsymbol{w} \\ w'_k \geq 0 \end{cases} .
\end{aligned} \tag{5.8}$$

At a first glance Equation 5.8 looks deceivingly simple (really, it is a one variable linear program only), however, one must keep in mind that for large problems this simple LP will have many constraints, and constraint generation may be required as a more efficient implementation.

The difference between the objective corresponding to the given solution pair $(\boldsymbol{w}, \boldsymbol{\lambda})$ and the objective value of Equation 5.8 is defined as score_plp():

$$\text{score\_plp}(\boldsymbol{\phi_k}) = \boldsymbol{\varphi}^T\boldsymbol{w} - \left(\varphi_k - \boldsymbol{\varphi}_{1:k-1}^T \boldsymbol{C}_B^{-1}\boldsymbol{c}_{B,k}w'_k\right)$$

Thus, each of the three scoring methods can determine with certainty whether the LP objective will be improved on or not.

5.2.3 Constructing the Basis Function Given a Candidate Domain

Finally, given a set of state variables, the task is to construct a non-linear[3] basis function with the set as its domain. An important issue in constructing this function may be the choice of representation (for compactness reasons); however, one of our assumptions was that the domain of the function remain small and the representation then can be just a look-up table. For binary variables the look-up table would be exponential in the number of variables making up the function domain.

[3] A linear function of these variables would just be vacuous.

We consider two methods of constructing basis functions on a given domain. The simpler one is an XOR[4] like function defined as

$$b_k(\boldsymbol{x}_k) = b_k(x_{k_1}, x_{k_2}, \ldots, x_{k_l}) = (-1)^{x_{k_1}}(-1)^{x_{k_2}} \cdots (-1)^{x_{k_l}}$$

The advantages offered by this method are:

- each new basis function is linearly independent of those in the current set of functions;
- the method is easy to implement and understand;
- basis functions are computationally easy to create.

Its downside is that it is not based on any insights or theory that would indicate production of quality basis functions. Therefore we are left with the task of testing the method experimentally with little known of what to expect.

The second method optimizes the basis function lookup table values for the given domain, to maximize score_dual(). Recall that the dual score of a function ϕ_k was partly[5] (and superficially) defined as

$$\max_{\boldsymbol{c}_{:,k}^T, \varphi_k} \boldsymbol{c}_{:,k}^T \boldsymbol{\lambda} - \varphi_k. \tag{5.9}$$

We can now put simplicity aside and reveal all details necessary to easily understand the optimization's solution. Let us suppose that the domain of ϕ_k is binary variables $\boldsymbol{x}_k$, and also that it is parameterized by the vector $\boldsymbol{\xi}$, denoted by $\phi_{k,\boldsymbol{\xi}}$. This implies a one-to-one correspondence between configurations of $\boldsymbol{x}_k$ and elements of $\boldsymbol{\xi}$. Thus the connection between

[4]The careful reader will have easily observed that the function thus presented relies on the fact that state variables $x_{k_1}, x_{k_2}, \ldots, x_{k_l}$ take 0/1 values. Although we can extend this type of basis function construction to multivalued variables using ideas from construction of Hadammard matrices, we only experimented with basis function construction on binary variables.

[5]Two identical linear optimizations have to be performed, one for a max and one for a min; we show only one here for obvious reasons.

$\boldsymbol{c}_{:,k}$, φ_k, and the parameters of $\boldsymbol{\phi}_{k,\boldsymbol{\xi}}$ becomes evident:

$$\boldsymbol{c}_{:,k} = \begin{bmatrix} (I-\gamma P^{a_1})\boldsymbol{\phi}_{k,\boldsymbol{\xi}} \\ (I-\gamma P^{a_2})\boldsymbol{\phi}_{k,\boldsymbol{\xi}} \\ \vdots \\ (I-\gamma P^{a_m})\boldsymbol{\phi}_{k,\boldsymbol{\xi}} \end{bmatrix} \tag{5.10}$$

$$= A \begin{bmatrix} \boldsymbol{\phi}_{k,\boldsymbol{\xi}} \\ \boldsymbol{\phi}_{k,\boldsymbol{\xi}} \\ \vdots \\ \boldsymbol{\phi}_{k,\boldsymbol{\xi}} \end{bmatrix} \tag{5.11}$$

$$\varphi_k = 2^{n-|\mathbf{Dom}(\boldsymbol{\phi}_{k,\boldsymbol{\xi}})|} \sum_{x_k} \phi_k(\boldsymbol{x}_k)$$

$$= 2^{n-|\mathbf{Dom}(\boldsymbol{\phi}_{k,\boldsymbol{\xi}})|} \sum_{j} \xi_j \tag{5.12}$$

where it is obvious what matrix A is.

Suppose that the current solution pair is $(\boldsymbol{w}, \boldsymbol{\lambda})$. There are two problems with the optimization in Equation 5.9. First we observe that the current dual solution vector $\boldsymbol{\lambda}$ has as many elements as there are states, hence making the optimization impractical. Luckily we are saved by the fact that most of the $\boldsymbol{\lambda}$ vector is filled with zeros. In fact—using previous notation—we can concentrate our attention on the non-zero part of the the vector: $\boldsymbol{\lambda}_B$. There are as many non-zero entries as there are active constraints in the LP, a number which is also equal to the number of primal variables (size of $\boldsymbol{w}$). We can use this fact right away in Equation 5.9, and also substitute the expressions from Equations 5.10 and 5.12 into Equation 5.9:

$$\max_{\boldsymbol{\xi}} \quad \sum_{i \in B} c_{i,k}\lambda_i - \varphi_k \tag{5.13}$$

$$\equiv \max_{\boldsymbol{\xi}} \quad \sum_{i \in B} c_{i,k}\lambda_i - 2^{n-|\mathbf{Dom}(\boldsymbol{\phi}_{k,\boldsymbol{\xi}})|} \sum_{j} \xi_j \tag{5.14}$$

$$\equiv \max_{\boldsymbol{\xi}} \quad \sum_{i \in B} A_{i,k}\xi_i\lambda_i - 2^{n-|\mathbf{Dom}(\boldsymbol{\phi}_{k,\boldsymbol{\xi}})|} \sum_{j} \xi_j \tag{5.15}$$

$$\equiv \max_{\boldsymbol{\xi}} \quad \boldsymbol{\tau}^T\boldsymbol{\xi} \tag{5.16}$$

The only step requiring clarification is from Equation 5.15 to the vector dot product of

Equation 5.16. This is possible simply by factoring out each ξ_i, thus obtaining its coefficient τ_i.

The second difficulty to overcome (and, instead of using Equation 5.9 we use Equation 5.16) is that, left this way, the variables can shoot to infinity; in other words, the optimization is unbounded. The typical solution is to constrain variables artificially to some reasonable values. For example, we could add the constraint that a norm of ξ equals 1:

$$\begin{aligned} \max_{\boldsymbol{\xi}} \quad & \boldsymbol{\tau}^T \boldsymbol{\xi_i} \\ \text{subject to} \quad & ||\boldsymbol{\xi}|| = 1 \end{aligned} \tag{5.17}$$

The choice of norm depends (from our point of view) partly on what effect it has on the optimization: does it make it impractical?—or leaves it unchanged. We consider three norms: L_1, L_2, and L_∞. Using an L_1 normalization yields a non-linear program of several variables and one constraint. A naïve linearization would end up with an exponential number of constraints. Using L_2 and L_∞ results in linear objectives with non-linear constraints, which cannot naively be made into linear programs. Considering that we might need to construct and score many basis functions, and therefore, computing one of these optimizations many times, the prospects of either an exponential number of constraints or, worse, non-linear optimizations is not very good. However, behind each of the three methods is hidden a simple linear time[6] operation.

Optimizing a Basis Function Whose Parameters are L_1 normalized

In this section we want to show that

$$\begin{aligned} \max_{\boldsymbol{\xi}} \quad & \boldsymbol{\tau}^T \boldsymbol{\xi} \\ \text{subject to} \quad & \sum_i |\xi_i| = 1 \end{aligned} \tag{5.18}$$

can be solved in linear time.

The optimal $\boldsymbol{\xi}^*$ is found by Algorithm 11:

Lemma 5.2.1 *A convex combination of a finite number of finite reals is at most as large as the largest of them.*

[6] Linear in the number of parameters of the basis function being created.

Algorithm 11 Basis function creation with L_1 constraint

1: $\boldsymbol{\xi}^* = \mathbf{0}$

2: $i^* = \arg\max_i |\tau_i|$

3: $\xi_{i^*} = \text{sign}(\tau_{i^*})$

where the *sign* function returns -1, 0, and 1 for respectively negative, zero, and positive arguments.

Proof. Let $\boldsymbol{x} \in \mathbb{R}^n$, and $-\infty < x_i < \infty$, $\forall i = 1 \ldots n$, and $\boldsymbol{w} \in \mathbb{R}^n$, $\sum_{i=1}^n w_i = 1$, and $w_i \geq 0$, $\forall i = 1 \ldots n$. We want to show that $\boldsymbol{x}^T \boldsymbol{w} \leq \max_i x_i$. Suppose the opposite holds: $\boldsymbol{x}^T \boldsymbol{w} > \max_i x_i$. Then, to avoid the trivial case, there must be at least two elements participating in the sum; i.e. $x_j w_j + x_k w_k > \max_i x_i$, and $\sum_{i, i \neq j, i \neq k} w_i = 0$, $w_j + w_k = 1$. We either have:

1. $w_j = 1$ and $w_k = 0$ which implies that $x_j > \max_i x_i$, a contradiction,[7] or
2. $w_j = 0$ and $w_k = 1$ which implies $x_k > \max_i x_i$, another contradiction, or
3. $0 < w_j < 1$ and $w_k = 1 - w_j$, which implies that $w_j(x_j - x_k) + x_k > \max_i x_i$. Now, either $x_j = x_k$, which leads to the contradiction that $x_k > \max_i x_i$, or $x_j < x_k$ which implies the contradiction that a number slightly smaller than x_k is greater than the max, or, finally, $x_j > x_k$, which leads to the contradiction that a number slightly larger than x_k but smaller than x_j is larger than the max.

Hence, our supposition that $\boldsymbol{x}^T \boldsymbol{w} > \max_i x_i$ holds must have been false. ■

Proposition 5.2.2 *Algorithm 11 solves the optimization of Equation 5.18 in $O(|\xi|)$ time.*

Proof. A quick glance at the algorithm suffices to be convinced that it is a linear time operation. To prove that it produces the optimal $\boldsymbol{\xi}$ we first observe that the L_1 normalization constraint imposed on the $\boldsymbol{\xi}$ variables implies also that $0 < |\xi_i| < 1$. Also, for any given $\boldsymbol{\xi}$, if any of its elements have negative signs, we can make them positive provided that we change the signs of the corresponding elements in $\boldsymbol{\tau}$, and the optimization will not change. Using Lemma 5.2.1 on this optimization proves the proposition. ■

[7] No other element can be greater than the max.

Optimizing a Basis Function Whose Parameters are L_2 normalized

In this section our goal is to show that

$$\begin{aligned} \max_{\boldsymbol{\xi}} \quad & \boldsymbol{\tau}^T \boldsymbol{\xi} \\ \text{subject to} \quad & \sqrt{\sum_i \xi_i^2} = 1 \end{aligned} \tag{5.19}$$

can be solved with a linear time operation.

Proposition 5.2.3

$$\boldsymbol{\xi}^* = \frac{\boldsymbol{\tau}}{||\boldsymbol{\tau}||_2}$$

Proof. Suppose the angle between $\boldsymbol{\tau}$ and $\boldsymbol{\xi}$ is θ. Then, by the definition of cosine,

$$\cos(\theta) = \frac{\boldsymbol{\tau}^T \boldsymbol{\xi}}{||\boldsymbol{\tau}||_2 \, ||\boldsymbol{\xi}||_2} \tag{5.20}$$

Helpful facts: the cosine function achieves a max when it is equal to 1; we have a constraint that $||\boldsymbol{\xi}||_2 = 1$. Substituting this in Equation 5.20 we get

$$\frac{\boldsymbol{\tau}^T \boldsymbol{\xi}}{||\boldsymbol{\tau}||_2} = 1$$

We conclude that the optimal value for $\boldsymbol{\xi}$ is $\frac{\boldsymbol{\tau}}{||\boldsymbol{\tau}||_2}$, since

$$\begin{aligned} 1 &= \frac{\boldsymbol{\tau}^T \boldsymbol{\xi}}{||\boldsymbol{\tau}||_2} && (5.21) \\ &= \frac{\boldsymbol{\tau}^T \boldsymbol{\tau}}{||\boldsymbol{\tau}||_2 ||\boldsymbol{\tau}||_2} && (5.22) \\ &= \frac{\boldsymbol{\tau}^T \boldsymbol{\tau}}{\sqrt{\boldsymbol{\tau}^T \boldsymbol{\tau}} \sqrt{\boldsymbol{\tau}^T \boldsymbol{\tau}}} && (5.23) \\ &= 1 && (5.24) \end{aligned}$$

■

Optimizing a Basis Function Whose Parameters are L_∞ normalized

In this section we want to show that

$$\begin{aligned} \max_{\boldsymbol{\xi}} \quad & \boldsymbol{\tau}^T \boldsymbol{\xi} \\ \text{subject to} \quad & \max_i |\xi_i| = 1 \end{aligned} \tag{5.25}$$

which can easily be converted into a linear program:

$$\begin{aligned} \max_{\boldsymbol{\xi}} \quad & \boldsymbol{\tau}^T \boldsymbol{\xi} \\ \text{subject to} \quad & -1 \leq \xi_i \leq 1 \quad \forall i \end{aligned} \tag{5.26}$$

is actually only a linear time procedure.

Proposition 5.2.4

$$\boldsymbol{\xi}_i^* = \mathit{sign}(\tau_i) \quad \forall i$$

Proof. Since the smallest and largest values of any element in $\boldsymbol{\xi}$ are bounded by -1 and 1 respectively, the largest objective can be obtained by simply changing the sign of negative entries in $\boldsymbol{\tau}$. ■

5.3 Experiments

Motivation, Goals, and Setup

We have proposed new methods for solution improvement by automatically constructing basis functions; however, it is difficult to ascertain with certainty the goodness of all the combinations of our proposed methods. We would like to be able to answer the following questions.

- Do our methods work at all?—that is, does at least the L_1 error decrease as the number of basis functions used increases?
- Do our methods decrease the other error measures (L_∞, Bellman error) as well?
- How rapidly, or according to what type of trend do the different errors decrease?

- To what extent is the decrease in different error measures correlated (helps in predicting more expensive error measures without calculating them)?
- Does the error indeed reduces to zero after adding enough basis functions?

In an effort to obtain more heterogeneous results we conducted experiments on a number of different problems: the `system administrator` domain (Appendix A.1), in three configurations: `cycle`, `3legs`, and `3loops`, the `resource` problem (Appendix A.3), the `robot` problem (Appendix A.2), and the `advisor` problem (Appendix A.4). To be able to answer the last question, we must be able to generate all the necessary basis functions to arrive at the exact solution. For large problems this can be prohibitively time consuming; therefore, we chose problem sizes that enabled us to finish the experiments in a reasonable time. Hence these experiments are not to showcase how large a problem we can efficiently obtain an approximation for; rather, they focus on providing answers to the posed questions. We used seven binary state variables for the `cycle`, `3legs`, and `3loops` problems, such that 128 basis functions suffice to solve the problem exactly. If we were to hand pick basis functions it would be as easy as matching each of the 128 states to one basis function, which returns a 0 for all other states but its own (a state indicator function)—but they would not have a compact representation, making them unsuitable but for small problems, and it would defeat the purpose. By any accounts these are small problems, which helps our investigation by allowing us to compute the exact solution and any error our approximations may incur.

Similarly, we set parameters for the `resource`, `robot`, and `advisor` to facilitate comparison of results; only the `advisor` problem requires 125 basis functions while the `resource` and `robot` problems require 128 basis functions.

All the problems used a discount factor of 0.95;[8] other problem parameters (e.g. transition probabilities, reward function) are set as described in the appendix. The code has been written in Matlab but we called the CPLEX 7.5 linear programming solver when ever needed. All simulations were run on 750MHz PCs.

Our methods offer 18 combinations for each problem:

- three ways of scoring basis functions, full LP (*score_flp*), partial LP (*score_plp*), and dual LP (*score_dual*);

[8] Other values for the discount factor had no impact on the conclusions we could draw from experiments.

- two ways of constructing basis functions, XOR (*xor*) and optimization with L_1 normalization (*optl1*);
- three methods for choosing the basis functions' domains, *sequential*, *lattice*, and *neighbor*.

Although we developed and showed earlier three ways of optimizing a new basis function, using L_1, L_2, and L_∞ normalization, we show only results done with an L_1 normalization; the other two normalization criteria did not offer significantly better (or worse) results, and we decided to omit their account.

Results and Discussion

The first series of simulations we performed aimed at comparing scoring methods. For the first three problems all 18 combinations were tried, while for the last three problems we excluded XOR basis function construction.

All figures are plots of an error measure on the y axis and number of used basis functions on the x axis. More pertinent details are given on each plot. For instance, the error measures shown are L_1 at the top, Bellman error in the middle, and L_∞ at the bottom. We have also prepared plots of the LP objective value against the number of basis functions but omit them for brevity, since their shape has been without exception almost identical to that of the L_1 plots. All the plots can be found in Appendix B.

Answers to some of the questions we posed are evident even after a quick glance at these figures. All plots show that the more basis functions are added, the more the error decreases. It is also clear that, as the last necessary basis function is added, the exact result is obtained and the error vanishes. In all cases the error seems to drop by a very significant amount after the first few basis functions are added (on average after about 15% of the total number of basis functions the L_∞ error dropped by about 83%). Among the six problems tried here, `resource` and `advisor` seem to have the more amenable structure for approximation, as they seem to require fewer basis functions than the other problems to achieve small error.

In all these plots the L_1 and L_∞ error plots are smoother and easier to see a trend in, while the Bellman error tends to jump all over the place. Even so, it is quite easy to see that the curves are highly correlated in all cases, especially the L_1 and L_∞ errors. In light of the fact that one is not able to minimize the L_∞ or Bellman error directly and efficiently, this

is an exciting result; it gives one hope that minimizing the L_1 error instead can result in a similar diminishing trend in the L_∞ error.

However, as we expected, there are significant differences among the different method combinations we investigate, and we would like to learn which is more advantageous to choose. We start by comparing scoring methods. While in this section we present only highlights, resulting plots comparing full LP, partial LP, and dual LP are shown in their entirety in Appendix B, pages 122–135.

As scoring methods go, we expected that *score_flp* would be more accurate in predicting the goodness of a basis function than *score_plp*, which in turn would be more accurate than *score_dual*. Although the expectation held, the difference in error measures is quite insignificant. In many cases using *score_dual* the error curve is very close to the that of *score_flp* and *score_plp*: this can be seen especially when optimizing the basis functions (*optl1*) and not using *xor* (e.g. Figure 5.1), on all problems but `robot` with *neighbor* domain selection (Figure 5.2). In general a marked—but not very big—difference between *score_dual* and the other two scoring methods can be seen when using the *xor* basis functions, on the `cycle`, `3legs`, and `3loops` problems especially when the domain was chosen with the *neighbor* method. This is good news, since we want to avoid the costlier methods and only use *score_dual*, which is a constant time operation[9] in the best case and linear in the worst (for *xor* functions).

Another point of interest is whether constructing basis functions via optimization or the *xor* method is better. To compare the two methods we show detailed plots in the appendix on pages 136–140. Having drawn a comparison on scoring methods, we continue with *score_dual* as the preferred procedure. The results are quite consistent across the three problems that we restrict our attention to in this comparison: `cycle`, `3legs`, and `3loops`. Slightly better performance is obtained by *optl1*, especially when domains are chosen via *lattice* and *neighbor* (e.g. Figure 5.3). When domains are chosen with *sequential* the curve shapes for *xor* and *optl1* are almost identical, as illustrated in Figure 5.4.

One last comparison we want to make is among domain choice methods. According to our previous results, we choose to show only the top performers and thus fix the scoring method to *score_dual* and basis function construction to *optl1*. Each graph on pages 141–143 in the appendix shows the error curves obtained by running the algorithms to completion (until the error vanished) on all six problems. Recall that of the three domain choosing

[9]The basis function construction produces the score as a side effect.

procedures, *neighbor* is the most aggressive in its progression to domains of many variables, while *sequential* is the most conservative. The two simulations which show the greatest difference are those solving the `cycle` and `3legs` problems. For instance, in Figure 5.5, the L_∞ error for *neighbor* drops the fastest, reaching about 16% of its initial value when 20 basis functions are used, compared to only 32% for *sequential* and *lattice*. However, on the `advisor` problem it is *sequential* which drops the error fastest, using only 6 basis functions compared to *neighbor* using 12 basis functions to achieve the same level of accuracy (Figure 5.6). Although *neighbor* does not perform best in all cases, it does seem to be the better method overall.

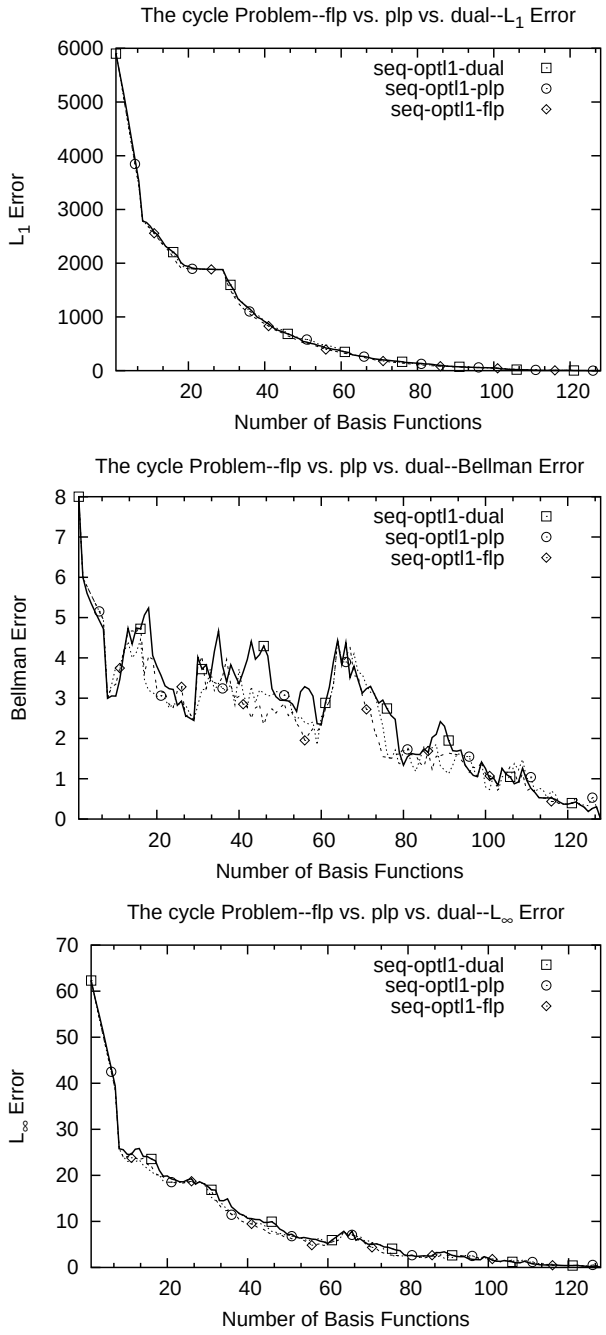


Figure 5.1: Example of good performance for the *score_dual* scoring method.

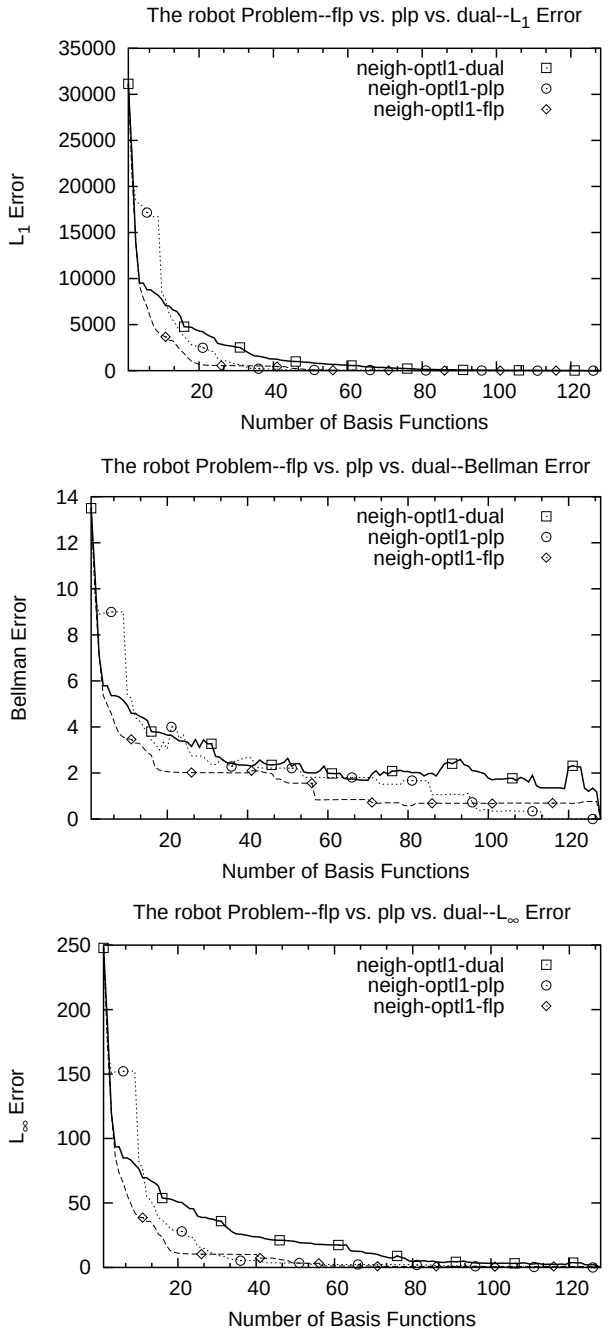


Figure 5.2: Example of slightly lower performance for the *score_dual* scoring method.

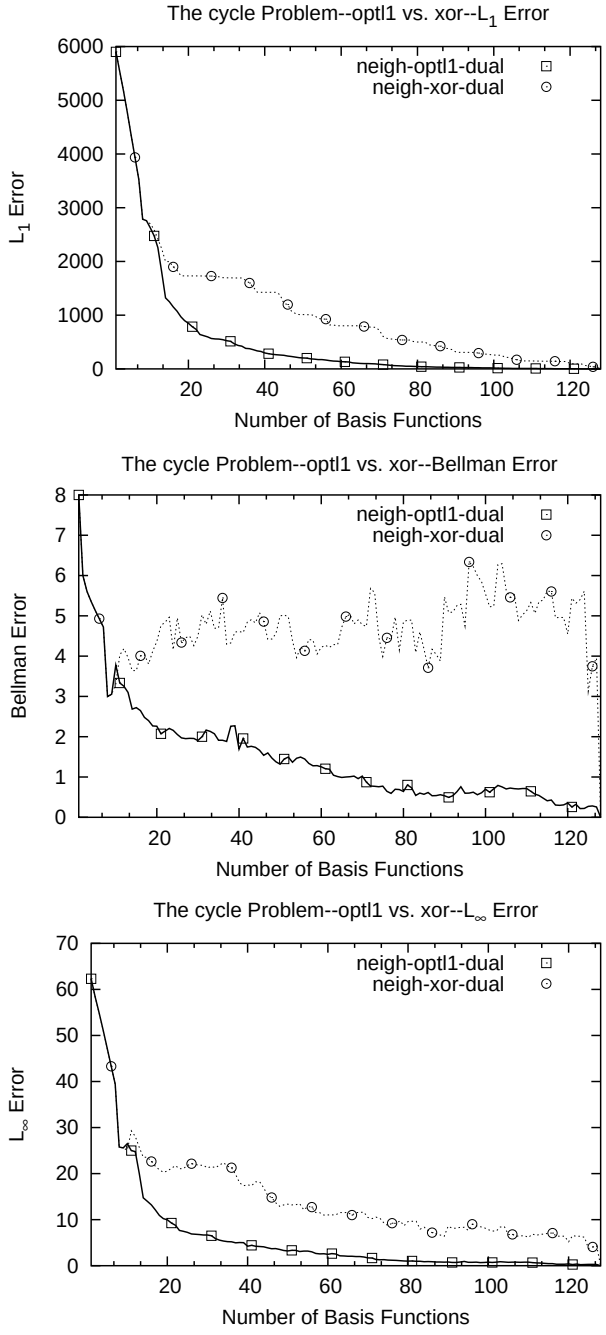


Figure 5.3: Example of lower performance when constructing a basis function with *xor* versus *optl1*.

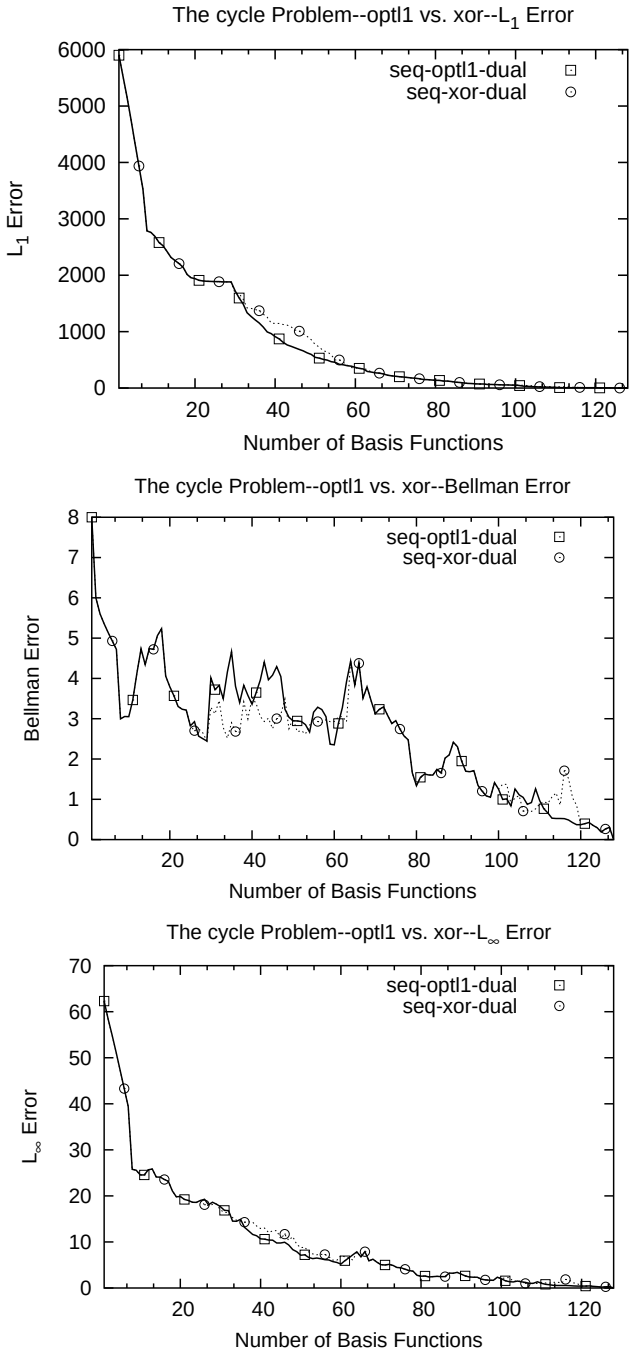


Figure 5.4: Example of similar performance for both *xor* and *optl1*.

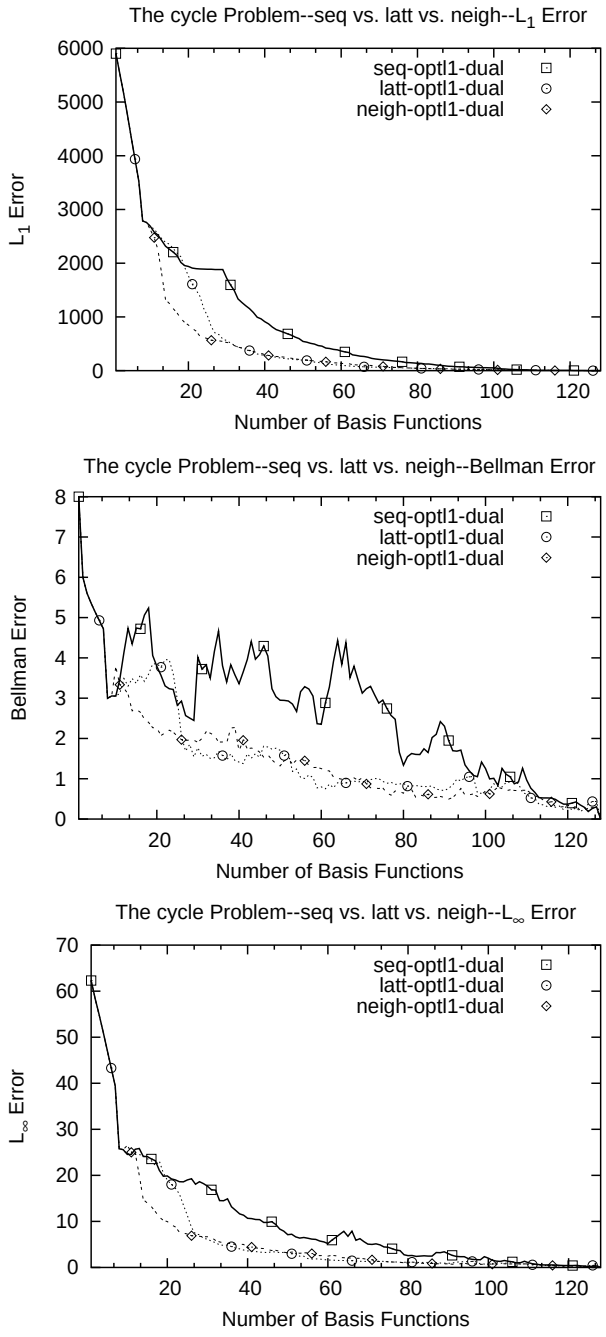


Figure 5.5: Example of good performance for the *neighbor* domain choosing method.

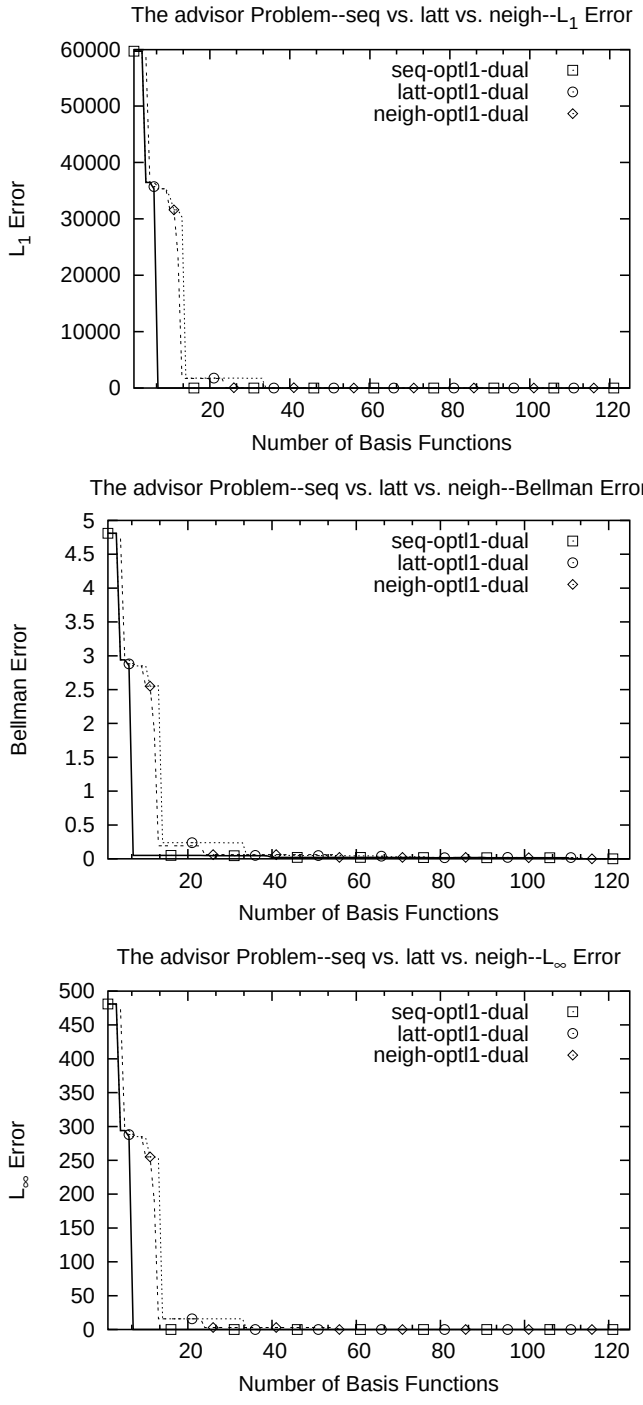


Figure 5.6: Example of good performance for the *sequential* domain choosing method.

Chapter 6

Conclusions

It seems that the big thorn in the side of much current research in several subfields of artificial intelligence is the inability to solve big, complex, real world problems. The desire to solve such problems, otherwise provably hard from a computational complexity standpoint, was the driving motivation for us. We feel that this thesis' contributions constitute progress in the right direction. Specifically, we believe that being able to compute the Bellman error without using brute force is progress, being able to compute an approximate solution in seven seconds where the state of the art algorithm solved it in seven hours is progress, as is the ability to improve an approximation error to the extent that one's computational resources allow.

We are aware that our algorithms do not have optimality theory behind them, and that this may be unpleasant for some, but we strived to provide convincing empirical evidence for the features we deemed interesting.

In summary, in Chapter 3 we provided a novel branch-and-bound algorithm for computing the exact Bellman error of approximate solutions in compactly represented MDPs, in Chapter 4 we developed algorithms based on the direct use of linear programming to approximately solve such MDPs given a set of basis functions to represent the solution, and in Chapter 5 we introduced a suite of methods for improving the accuracy of the approximate solution by expanding the set of current basis functions.

Let us mention some attractive features of our methods:

- the ability to compute the Bellman error without resorting to brute force;
- the ability to solve factored MDPs while minimizing the Bellman error without a "default action" assumption;
- the ability to drastically reduce the computation time of approximately solving factored MDPs;
- the ability to improve the current approximation, in effect making it an any-time algorithm;
- a principled way to choose new basis functions such that the error is reduced;
- automatic and efficient construction of basis functions;
- several configuration choices, allowing the designer to custom fit the algorithm to the problem at hand; alas we cannot claim that the combination we found to work well

on the six problems we experimented with will work best on all problems.

There is, however, room for improvement, and we list some possible future research directions. For every new basis function that gets added to the current set a search is performed over many domains. The domain whose constructed basis function has the highest score is then chosen. The main source of inefficiency here is the large number of domains to consider at every iteration and calculate a score for, even though computing the score is fast. Several heuristic based solutions for this impediment can be further proposed. For instance, since for many domains a score is computed time and again, one can cache and re-use older scores. These cached scores would probably be adequate if new scores are re-computed often enough while the overall algorithm time can be shortened significantly. Another improvement could be implemented by adding more than one basis functions to the current set at each iteration. The choice can be made considering either cached or current scores.

We have also experimented with and obtained very preliminary results from a technique which eliminates the explicit search over basis function domains. The method uses the fact that a primal LP column is a dual LP row, and choosing a basis function is for the dual much like a new constraint is generated when solving the primal LP. We hope that this method can provide further speed improvements over the methods we introduced, allowing us to solve even larger problems.

If one sets an upper limit on the number of variables allowed to make up the domain of basis functions, then choosing the variables may be done using a procedure based on variable elimination. For any given variable elimination order, intermediate factors are formed whose size (in number of variables) determines the efficiency of the variable elimination procedure. Since this efficiency has a direct impact on the search for the maximum violated constraint, one can choose domains for new basis functions such that they are at most as large as the upper limit, and that no intermediate factor has size greater than what can be handled safely with available computing.

Appendix A

Problem Description

A.1 The system administrator Domain

This artificial problem domain is due to Guestrin *et al.* [27] and its appeal for experimentation can be traced to three features: inherent structure, flexibility, and scalability. The structure that each instance of this problem exhibits lends itself very well to compact representation, and therefore to the algorithms proposed here, just as they do to the algorithms of Guestrin et al. One can easily create problem instances of various configurations, allowing great flexibility in choosing how state variables influence each other, and therefore how easy or difficult the instance is. Finally, given a fixed network configuration, one can easily create instances as small or large as desired for experimentation.

The system administrator domain is described as a number of computers that are connected in a network, in a particular configuration, which influences directly the dynamics of the system. We experiment with instances in which one of the computers is designated as server, while the rest are clients. The server is presumed somewhat more important in a network, and this fact is reflected in both the transition probabilities and the reward function.

The state of the system is represented as a vector of binary variables, one for each machine in the network. The value of a variable indicates whether the respective machine is up (1) or down (0). At each stage a computer can go down according to a probability which is increased if its neighbors (adjacent machines in the network) were down in the previous stage. Each machine that is up contributes 1 to the reward, except for the server, who contributes 2. At each stage the system administrator can take the action of either rebooting one of the machines, in which case it may come up, or not reboot any machine. Machines also may spontaneously come up with a small probability. The actual transition probabilities we used were:

- $P(x'_i = 1 | x_i, \mathit{parent}(x_i), a = i) = 0.95$
- $P(x'_i = 1 | x_i = 1, \mathit{parent}(x_i) = 1, a \neq i) = 0.9$
- $P(x'_i = 1 | x_i = 1, \mathit{parent}(x_i) = 0, a \neq i) = 0.67$
- $P(x'_i = 1 | x_i = 0, \mathit{parent}(x_i), a \neq i) = 0.01$

For all instances of this domain the discount factor we used was $\gamma = 0.95$.

We experimented with a variety of network configurations, and show here results on three particular layouts. In the first one, the machines are networked in a loop (as shown in Figure A.1), and influence each other only in one direction: we names this a `cycle` configuration.[1] The second layout has a three legged star shape with the server in the middle, as shown in Figure A.2. The third configuration is similar to the star configuration, except that each of the outward legs forms itself a loop (Figure A.3).

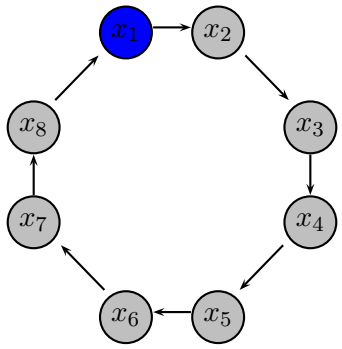


Figure A.1: Graphical depiction of the `cycle` configuration for the system administration problem.

[1] We will also call this the `cycle` problem, meaning really the `system administrator` domain with a `cycle` configuration.

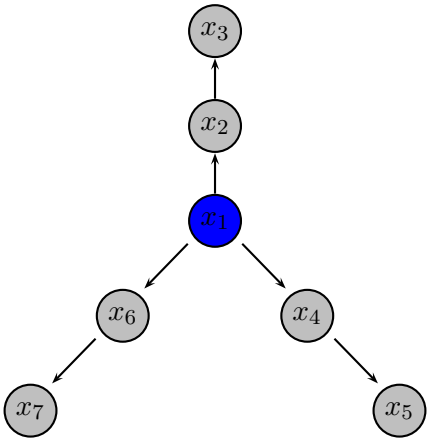


Figure A.2: Graphical depiction of the `3legs` configuration for the system administration problem.

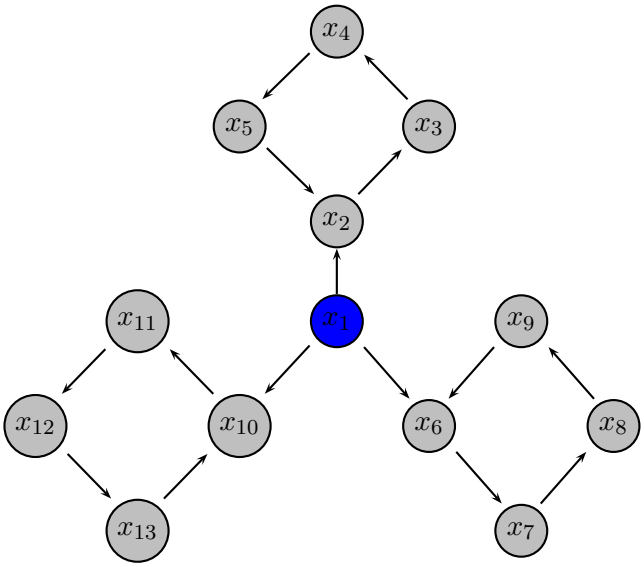


Figure A.3: Graphical depiction of the `3loops` configuration for the system administration problem.

A.2 The robot Problem

This problem is due to Craig Boutilier [9]. The domain envisions a robot whose task is to deliver mail and coffee to a number of users. There is a set of n users $\{U_1, \ldots, U_n\}$, and a set of locations $\{C, M, H, O_1, \ldots, O_n\}$. We model the problem as follows.
State variables:

- $Loc \in \{C, M, O_1, \ldots, O_n\}$, a multi-value variable for encoding locations;
- $WC(i) \in \{0, 1\}$ —user $i = 1, \ldots, n$ wants coffee;
- $MW(i) \in \{0, 1\}$ —mail is waiting for user i;
- $HC \in \{0, 1\}$ —robot has coffee;
- $HM(i) \in \{0, 1\}$ —robot has mail for user i;
- $BatOK \in \{0, 1\}$ —robot battery status.

Actions:

- $move(H, l)$ —move from home to any other location l;
- $move(l, H)$ —move from any other location l back home;
- $GetCof$ —get coffee;
- $GetMail(i)$ —get mail for user i;
- $DelCof(i)$ —deliver coffee to user i;
- $DelMail(i)$ —deliver mail to user i;
- $ChargeBat$ —charge battery if $BatOK$ is 0.

Dynamics:

1. $move(l_j, l_k)$: affects Loc, $BatOK$, $MW(i)$, and $WC(i)$.

Loc' depends on $BatOK$ and Loc:

$P(Loc' = l_k \mid Loc = l_j, BatOK = 1) = p\_succ$ the move from l_j to $l_k succeeded, and l_k \neq l_j$;
$P(Loc' = l_j \mid Loc = l_j, BatOK = 1) = 1 - p\_succ$ move did not succeed, so the robot's stuck in the san
$P(Loc' = l_k \mid Loc = l_k, BatOK = 0) = 1$ i.e. robot did not move if battery is drained;
$P(Loc' = lj \mid Loc = lj) = 1,$ for any $lj \neq l1$;

where we used $p\_succ = 0.9$ as the probability that a move succeeds.

$BatOK'$ depends on $BatOK$: $P(BatOK' = 1|BatOK) = 1 - p^d rain$, where p_drain and was set to 0.1 to avoid causing resource contention.

$MW(i)'$ depends on $MW(i)$:

$$P(MW(i)' = 1 \mid MW(i) = 1) = 1 - p\_wait\_mail_i$$
$$P(MW(i)' = 1 \mid MW(i) = 0) = p\_wait\_mail_i$$

where we used $p\_wait\_mail_i = 0.2$ as the probability that mail is waiting at any stage for user i.

$WC(i)'$ depends on $WC(i)$:

$$P(WC(i)' = 1 \mid WC(i) = 1) = 1$$
$$P(WC(i)' = 1 \mid WC(i) = 0) = p\_want\_cof_i$$

where $p\_want\_cof_i$ is the probability that at any instant user i requests coffee and was set in our experiments to 0.25;

2. $DelCof(i)$: affects $HC, WC(i), BatOK, MW(j)$ (for all j), $WC(j)$ (for all $j \neq i$)

   HC' depends on HC and $BatOK$:

$$P(HC' = 1 \mid BatOK = 1) = 0$$
$$P(HC' = 1 \mid HC, BatOK = 0) = 1$$
$$P(HC' = 1 \mid HC = 0, BatOK = 0) = 0$$

$WC(i)'$ depends on HC, Loc, $WC(i)$, $BatOK$

$$\begin{aligned} P(WC(i)' &\mid HC, Loc = O_i, WC(i), BatOK) = 0.2 \text{ (or some failure prob)} \\ P(WC(i)' = 1 &\mid HC = 0, Loc \neq O_i, WC(i) = 1, BatOK = 1) = 1.0 \\ P(WC(i)' = 1 &\mid WC(i), BatOK = 0) = 1.0 \\ P(WC(i)' = 1 &\mid WC(i) = 0) = p\_want\_cof_i \end{aligned}$$

The parameters for $MW(j)$, $WC(j)$, and $BatOK$ are the same as for the $move(l_j, l_k)$ action, for all j.

3. $DelMail(i)$: affects $HM(i)$, $MW(i)$, $BatOK$, $WC(j)$ (for all j), $MW(j)$ (for all $j \neq i$)

   $HM(i)'$ depends on $HM(i)$,$BatOK$:

$$\begin{aligned} P(HM(i)' = 1 &\mid BatOK = 1) = 0 \\ P(HM(i)' = 1 &\mid HM(i) = 1, BatOK = 1) = 1 \\ P(HM(i)' = 1 &\mid HM(i) = 0, BatOK = 1) = 0 \end{aligned}$$

   $MW(i)'$ depends on $HM(i)$, Loc, $MW(i)$, B$atOK$

$$\begin{aligned} P(MW(i)' = 1 &\mid HM(i) = 1, Loc = O_i, BatOK = 1) = 0 \text{ (mail always succeeds)} \\ P(MW(i)' = 1 &\mid HM(i) = 0, Loc \neq O_i, MW(i) = 1, BatOK = 1) = 1.0 \\ P(MW(i)' = 1 &\mid MW(i) = 1, BatOK = 0) = 1.0 \\ P(MW(i)' = 1 &\mid MW(i) = 0) = p\_mail_i \end{aligned}$$

   The parameters for $MW(j)$ (for all $j \neq i$, $WC(j)$ (for all j), and $BatOK$ are the same as for the $move(l_j, l_k)$ action.

4. $GetCof$: affects HC, $WC(j)$, $BatOK$, $MW(j)$

   HC' depends on HC,$BatOK$,Loc:

$$\begin{aligned} P(HC' = 1 &\mid HC = 0, Loc = C, BatOK = 1) = 0.9 \text{ (or some pickupcoffee success prob)} \\ P(HC' = 1 &\mid HC = 0, Loc \neq C, BatOK = 0) = 0 \\ P(HC' = 1 &\mid HC = 1) = 1 \end{aligned}$$

Note: the robot can hold only one cup of coffee.

The parameters for $MW(j)$ (for all $j \neq i$, $WC(j)$ (for all j), and $BatOK$ are the same as for the $move(l_j, l_k)$ action.

5. $GetMail(i)$: affects $HM(i)$, $WC(j)$, $BatOK$, $MW(j)$

   $HM(i)'$ depends on $HM(i)$,$BatOK$,Loc:

   $P(HM(i)' = 1 \mid HM(i) = 0, Loc = M, BatOK = 1) = 0.9$ (or some pickupmail success prob)
   $P(HM(i)' = 1 \mid HM(i) = 0, Loc \neq M, BatOK = 0) = 0$
   $P(HM(i)' = 1 \mid HM(i) = 1) = 1$

   Note: we assume that the robot can carry all existent mail.

   The parameters for $MW(j)$ (for all $j \neq i$, $WC(j)$ (for all j), and $BatOK$ are the same as for the $move(l_j, l_k)$ action.

6. $ChargeBat$: affects $HM(i)$, $MW(i)$, $WC(i)$, HC, $BatOK$

   All variables above become false with probability 1 (and have no parents), except for $BatOK$, which becomes true with probability 1.

   Note: the robot can charge battery wherever it is.

The reward model:

- value of delivering coffee to user i is VC_i;
- value of delivering mail to user i is VM_i;
- reward $R(DelCof(i), HC = 1, Loc = O_i, WC(i) = 1, BatOK = 1) = p\_succ_i * VC_i$;
- reward $R(DelCof(i), \cdot) = 0$, i.e. under any other conditions the reward of coffee delivery is 0;
- reward $R(DelMail(i), HM(i) = 1, Loc = O_i, MW(i) = 1, BatOK = 1) = p\_succ_i * VM_i$;

- reward $R(DelMail(i), \cdot) = 0$, i.e. under any other conditions, the reward for mail delivery is 0.

To summarize, for the `robot` experiments we set parameters to the following values:

Parameter	**Value**
p_succ	0.9
p_drain	0.1
p_wait_mail	0.2
p_want_cof	0.25
p_get_mail	0.95
p_get_cof	0.9
p_del_mail	0.95
p_del_cof	0.9
$R(DelMail(i), HM(i) = 1, Loc = O_i, MW(i) = 1, BatOK = 1)$	10
$R(DelCof(i), HC = 1, Loc = O_i, WC(i) = 1, BatOK = 1)$	15

A.3 The resource Problem

This is a resource allocation problem, and is due to Craig Boutilier [8]. There is a set of n tasks which, in order to finish, need some allocation of some of the m available resources. We choose the straight forward state representation as $n + m$ binary variables: $T_1, \ldots, T_n$ are task variables, each taking a value of 1 if the task is currently active (i.e. in need of resources), and 0 otherwise; similarly, $Re_1, \ldots, Re_m$ are resource variables, each taking value 1 if the resource is available and 0 otherwise.

Allocating resources corresponds to actions, therefore there are $(n + 1)^m$ number of actions (each resource can either be withhold of assigned to one of the tasks). To encode actions as binary variables we assume the number of tasks is: $n = 2^l - 1$. Hence, for a given resource, there are l action variables that will take 2^l configurations: one configuration corresponding to the resource not being assigned, and $2^l - 1$ configurations corresponding to respective assignments to tasks. There are therefore ml number of action variables $A_{11}, \ldots, A_{ml}$.

The resource problem dynamics Generally speaking, tasks become periodically active. A task can finish spontaneously for no reason, or because enough resources have successfully been allocated to it. Temporally, variable T_i' depends on its state in the previous stage, T_i, on all action variables, A_{jk}, as well as on all resource variables Re_j (Figure A.4). Similarly, variable Re_j' depends on its prior status variable Re_j and on its corresponding action variables $A_{j1}, \ldots, A_{jl}$ (Figure A.5).

More specifically:

- $P(T_i' = 1 | T_i = 0, \ldots) = p\_occur_i$. The probability of any task variable becoming active is p_occur_i
- $$P(T_i' = 0 | T_i = 1, \ldots) = \begin{cases} (1 - p\_failsolve)^{num\_resources_i} & \text{if } num\_resources_i > 0 \\ p\_disappear_i & \text{otherwise} \end{cases}$$

  where $num\_resources_i$ is the number of resources allocated to task i (basic noisy OR model), and $p\_disappear_i$ is the probability that task i disappears on its own.

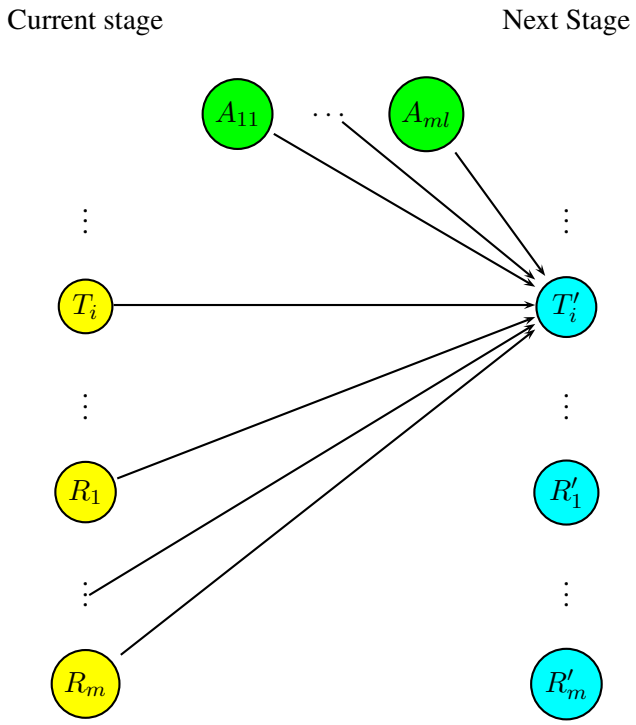


Figure A.4: Partial view of the DBN for the resource problem: dependence of task variable T'_i on action variables and on its prior status.

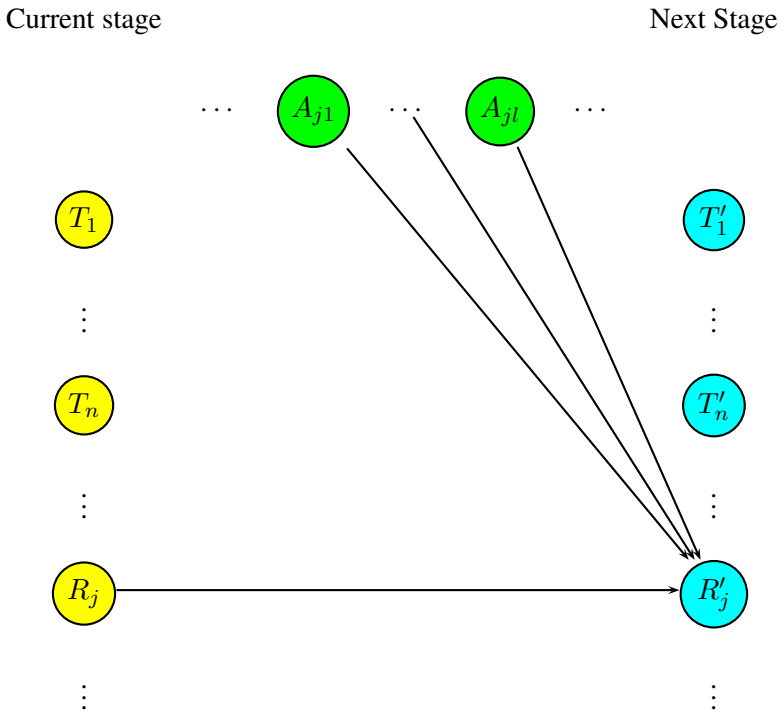


Figure A.5: Partial view of the DBN for the resource problem: dependence of task variable R'_j on corresponding action variables and on its prior status. Note, we omitted all other links from the graph (e.g. those pointing to T'_i).

If the task succeeds (e.g. when a number of resources have been allocated to it), then reward R_i is received, otherwise, no reward is given. Or:

$$R(T_i = 1, A_{ij}) = (1 - p\_failsolve^{num\_resources_i}) * V_i$$

for any action A_{ij} which applies more than zero resources to task i, and zero reward for any other action or if no resources are allocated to the task. V_i is a scalar, the value of an objective being accomplished.

The status of each resource Re_j depends on its action variables and its prior status:

$$\begin{aligned} P(Re'_j = 1 \mid Re_j = 1, A_{ij}) &= \begin{cases} p\_exhaust_j & \text{if the action } A_{ij} \text{ applies } Re_j \text{ to task } i \\ 1 & \text{if } Re_j \text{ is held back} \end{cases} \\ P(Re'_j = 1 \mid Re_j = 0, A_{ij}) &= p\_revive_j \end{aligned}$$

We set the numerical values of this problem's parameters in an effort to make it interesting: a scattering of high value and low value objectives (for accomplished tasks). We also set some tasks with reasonably high "disappear" probabilities, so that there is urgency to apply resources to them. And we also set some high value tasks with reasonably high "pop up" probability: this means that some resources might be held back in reserve in anticipation of the occurrence of high value tasks. Thus we can make this problem interesting with only one or two resources. We tried to keep $p\_exhaust_j$ fairly high to ensure there really is resource contention.

A.4 The advisor Problem

General Description This problem is due to Judy Goldsmith and Pascal Poupart [24]. It is an artificially constructed problem based on a real world domain from the Master's program in computer science at the University of Kentucky.[2] The domain is is inspired from the problem faced by an academic advisor whose job is to help students plan their course choices for one or more of the upcoming semester, after discussing their background and considering their transcripts. The advisor must take into consideration the student's supporting knowledge of relevant pre-requisite courses, ability to learn, utility of passing a certain course, and some hard constraints such as overlapping course schedule, periodic absence from campus, etc. Thus, course inter-dependency and students' preferences are important factors in the decision making task.

The problem is modeled as an MDP as follows. The (simplified) objective is the maximization of number of credits earned by the student over time. A number of credits is allocated to each course which is earned by the student should a passing mark be obtained. Possible grades are A, B, and C for passing, E for failing, and none for never taking the course. Courses taken repeatedly are only considered if the last grade obtained was a passing mark. The grade a student is likely to get in a course depends probabilistically on the grades obtained for passed pre-requisite courses. The transition probabilities are set as show in the next section. A student is assumed to be able to take up to four courses a year. A final note concerning hard pre-requisite constraints is that they are encoded in such way that invalid actions are given a reward of $-\infty$. Thus, these actions are numerically disqualified automatically if any other choice of actions exist, which offer a finite reward. We assume that in any MDP state there exists at least one finite-reward action.

State variables Let us assume there are n available courses. State variables encode whether any of the available courses were taken and the grade obtained. That is: $c_i \in \{A, B, C, E, none\},\ i = 1, \ldots, n$.

Actions There is one action for each valid choice of courses to take, from no courses taken up to the maximum number of courses allowed per semester.

[2] Simplified in various ways.

The Transition Model The value of a state variable in the next stage either has probabilistic dependency on the values of state variables for pre-requisite courses in the current stage, if there were any pre-requisites, or it has no dependencies if the course has no pre-requisites.

The following table shows the pre-requisites used in our experiments.

Course	**Pre-requisites**
c_1	$none$
c_2	c_1
c_3	c_2
c_4	$none$
c_5	$c_1,\ c_3$
c_6	$none$
c_7	c_5
c_8	c_2
c_9	$c_1,\ c_4$
c_{10}	$none$

Below we give the actual transition probabilities. The conditional probability table (CPT) for the "no dependency" case was:

$$\begin{aligned} P(c_i' = A \quad | \quad take\_course = i) &= 0.5 \\ P(c_i' = B \quad | \quad take\_course = i) &= 0.3 \\ P(c_i' = C \quad | \quad take\_course = i) &= 0.15 \\ P(c_i' = E \quad | \quad take\_course = i) &= 0.05 \\ P(c_i' = none \quad | \quad take\_course = i) &= 0 \end{aligned}$$

for $i = 1, \ldots, n$.

For dependency on one pre-requisite.

- The grade obtained for the pre-requisite course was A.

$$\begin{aligned} P(c_i' = A \quad | \quad take\_course = i, c_j = A) &= 0.70 \\ P(c_i' = B \quad | \quad take\_course = i, c_j = A) &= 0.20 \\ P(c_i' = C \quad | \quad take\_course = i, c_j = A) &= 0.08 \\ P(c_i' = E \quad | \quad take\_course = i, c_j = A) &= 0.02 \\ P(c_i' = none \quad | \quad take\_course = i, c_j = A) &= 0 \end{aligned}$$

- The grade obtained for the pre-requisite course was B.

$$\begin{aligned} P(c'_i = A \mid take\_course = i, c_j = B) &= 0.45 \\ P(c'_i = B \mid take\_course = i, c_j = B) &= 0.35 \\ P(c'_i = C \mid take\_course = i, c_j = B) &= 0.15 \\ P(c'_i = E \mid take\_course = i, c_j = B) &= 0.05 \\ P(c'_i = none \mid take\_course = i, c_j = B) &= 0 \end{aligned}$$

- The grade obtained for the pre-requisite course was C.

$$\begin{aligned} P(c'_i = A \mid take\_course = i, c_j = C) &= 0.25 \\ P(c'_i = B \mid take\_course = i, c_j = C) &= 0.25 \\ P(c'_i = C \mid take\_course = i, c_j = C) &= 0.35 \\ P(c'_i = E \mid take\_course = i, c_j = C) &= 0.15 \\ P(c'_i = none \mid take\_course = i, c_j = C) &= 0 \end{aligned}$$

- The grade obtained for the pre-requisite course was E.

$$\begin{aligned} P(c'_i = A \mid take\_course = i, c_j = E) &= 0.10 \\ P(c'_i = B \mid take\_course = i, c_j = E) &= 0.20 \\ P(c'_i = C \mid take\_course = i, c_j = E) &= 0.30 \\ P(c'_i = E \mid take\_course = i, c_j = E) &= 0.40 \\ P(c'_i = none \mid take\_course = i, c_j = E) &= 0 \end{aligned}$$

- The pre-requisite was not taken, therefore we encode it as $none$.

$$\begin{aligned} P(c'_i = A \mid take\_course = i, c_j = none) &= 0.20 \\ P(c'_i = B \mid take\_course = i, c_j = none) &= 0.25 \\ P(c'_i = C \mid take\_course = i, c_j = none) &= 0.30 \\ P(c'_i = E \mid take\_course = i, c_j = none) &= 0.25 \\ P(c'_i = none \mid take\_course = i, c_j = none) &= 0 \end{aligned}$$

Where $i = 1, \ldots, n$, and c_j is the pre-requisite course variable.
For dependency on two pre-requisites. One pre-requisite mark is A.

- The grades obtained for pre-requisite courses were A for the first and A for the second.

$$\begin{aligned}
P(c'_i = A \quad &| \quad take_course = i, c_j = A, c_k = A) = 0.8 \\
P(c'_i = B \quad &| \quad take_course = i, c_j = A, c_k = A) = 0.15 \\
P(c'_i = C \quad &| \quad take_course = i, c_j = A, c_k = A) = 0.04 \\
P(c'_i = E \quad &| \quad take_course = i, c_j = A, c_k = A) = 0.01 \\
P(c'_i = none \quad &| \quad take_course = i, c_j = A, c_k = A) = 0
\end{aligned}$$

- The grades obtained for pre-requisite courses were A for the first and B for the second.

$$\begin{aligned}
P(c'_i = A \quad &| \quad take_course = i, c_j = A, c_k = B) = 0.65 \\
P(c'_i = B \quad &| \quad take_course = i, c_j = A, c_k = B) = 0.25 \\
P(c'_i = C \quad &| \quad take_course = i, c_j = A, c_k = B) = 0.08 \\
P(c'_i = E \quad &| \quad take_course = i, c_j = A, c_k = B) = 0.02 \\
P(c'_i = none \quad &| \quad take_course = i, c_j = A, c_k = B) = 0
\end{aligned}$$

- The grades obtained for pre-requisite courses were A for the first and C for the second.

$$\begin{aligned}
P(c'_i = A \quad &| \quad take_course = i, c_j = A, c_k = C) = 0.5 \\
P(c'_i = B \quad &| \quad take_course = i, c_j = A, c_k = C) = 0.3 \\
P(c'_i = C \quad &| \quad take_course = i, c_j = A, c_k = C) = 0.15 \\
P(c'_i = E \quad &| \quad take_course = i, c_j = A, c_k = C) = 0.05 \\
P(c'_i = none \quad &| \quad take_course = i, c_j = A, c_k = C) = 0
\end{aligned}$$

- The grades obtained for pre-requisite courses were A for the first and E for the second.

$$\begin{aligned}
P(c'_i = A \quad &| \quad take_course = i, c_j = A, c_k = E) = 0.3 \\
P(c'_i = B \quad &| \quad take_course = i, c_j = A, c_k = E) = 0.3 \\
P(c'_i = C \quad &| \quad take_course = i, c_j = A, c_k = E) = 0.2 \\
P(c'_i = E \quad &| \quad take_course = i, c_j = A, c_k = E) = 0.2 \\
P(c'_i = none \quad &| \quad take_course = i, c_j = A, c_k = E) = 0
\end{aligned}$$

- The grades obtained for pre-requisite courses were A for the first and $none$ for the second which was not taken.

$$\begin{aligned} P(c_i' = A \quad | \quad take\_course = i, c_j = A, c_k = none) &= 0.4 \\ P(c_i' = B \quad | \quad take\_course = i, c_j = A, c_k = none) &= 0.3 \\ P(c_i' = C \quad | \quad take\_course = i, c_j = A, c_k = none) &= 0.2 \\ P(c_i' = E \quad | \quad take\_course = i, c_j = A, c_k = none) &= 0.1 \\ P(c_i' = none \quad | \quad take\_course = i, c_j = A, c_k = none) &= 0 \end{aligned}$$

One pre-requisite mark is B.

- The grades obtained for pre-requisite courses were B for the first and A for the second.

$$\begin{aligned} P(c_i' = A \quad | \quad take\_course = i, c_j = B, c_k = A) &= 0.65 \\ P(c_i' = B \quad | \quad take\_course = i, c_j = B, c_k = A) &= 0.25 \\ P(c_i' = C \quad | \quad take\_course = i, c_j = B, c_k = A) &= 0.08 \\ P(c_i' = E \quad | \quad take\_course = i, c_j = B, c_k = A) &= 0.02 \\ P(c_i' = none \quad | \quad take\_course = i, c_j = B, c_k = A) &= 0 \end{aligned}$$

- The grades obtained for pre-requisite courses were B for the first and B for the second.

$$\begin{aligned} P(c_i' = A \quad | \quad take\_course = i, c_j = B, c_k = B) &= 0.40 \\ P(c_i' = B \quad | \quad take\_course = i, c_j = B, c_k = B) &= 0.50 \\ P(c_i' = C \quad | \quad take\_course = i, c_j = B, c_k = B) &= 0.07 \\ P(c_i' = E \quad | \quad take\_course = i, c_j = B, c_k = B) &= 0.03 \\ P(c_i' = none \quad | \quad take\_course = i, c_j = B, c_k = B) &= 0 \end{aligned}$$

- The grades obtained for pre-requisite courses were B for the first and C for the second.

$$\begin{aligned} P(c_i' = A \quad | \quad take\_course = i, c_j = B, c_k = C) &= 0.20 \\ P(c_i' = B \quad | \quad take\_course = i, c_j = B, c_k = C) &= 0.35 \\ P(c_i' = C \quad | \quad take\_course = i, c_j = B, c_k = C) &= 0.35 \\ P(c_i' = E \quad | \quad take\_course = i, c_j = B, c_k = C) &= 0.10 \\ P(c_i' = none \quad | \quad take\_course = i, c_j = B, c_k = C) &= 0 \end{aligned}$$

- The grades obtained for pre-requisite courses were B for the first and E for the second.

$$\begin{aligned} P(c'_i = A \quad | \quad take\_course = i, c_j = B, c_k = E) &= 0.20 \\ P(c'_i = B \quad | \quad take\_course = i, c_j = B, c_k = E) &= 0.25 \\ P(c'_i = C \quad | \quad take\_course = i, c_j = B, c_k = E) &= 0.30 \\ P(c'_i = E \quad | \quad take\_course = i, c_j = B, c_k = E) &= 0.20 \\ P(c'_i = none \quad | \quad take\_course = i, c_j = B, c_k = E) &= 0 \end{aligned}$$

- The grades obtained for pre-requisite courses were B for the first and $none$ for the second which was not taken.

$$\begin{aligned} P(c'_i = A \quad | \quad take\_course = i, c_j = B, c_k = none) &= 0.30 \\ P(c'_i = B \quad | \quad take\_course = i, c_j = B, c_k = none) &= 0.35 \\ P(c'_i = C \quad | \quad take\_course = i, c_j = B, c_k = none) &= 0.25 \\ P(c'_i = E \quad | \quad take\_course = i, c_j = B, c_k = none) &= 0.10 \\ P(c'_i = none \quad | \quad take\_course = i, c_j = B, c_k = none) &= 0 \end{aligned}$$

One pre-requisite mark is C.

- The grades obtained for pre-requisite courses were C for the first and A for the second.

$$\begin{aligned} P(c'_i = A \quad | \quad take\_course = i, c_j = C, c_k = A) &= 0.50 \\ P(c'_i = B \quad | \quad take\_course = i, c_j = C, c_k = A) &= 0.30 \\ P(c'_i = C \quad | \quad take\_course = i, c_j = C, c_k = A) &= 0.15 \\ P(c'_i = E \quad | \quad take\_course = i, c_j = C, c_k = A) &= 0.05 \\ P(c'_i = none \quad | \quad take\_course = i, c_j = C, c_k = A) &= 0 \end{aligned}$$

- The grades obtained for pre-requisite courses were C for the first and B for the second.

$$\begin{aligned} P(c'_i = A \quad | \quad take\_course = i, c_j = C, c_k = B) &= 0.20 \\ P(c'_i = B \quad | \quad take\_course = i, c_j = C, c_k = B) &= 0.35 \\ P(c'_i = C \quad | \quad take\_course = i, c_j = C, c_k = B) &= 0.35 \\ P(c'_i = E \quad | \quad take\_course = i, c_j = C, c_k = B) &= 0.10 \\ P(c'_i = none \quad | \quad take\_course = i, c_j = C, c_k = B) &= 0 \end{aligned}$$

- The grades obtained for pre-requisite courses were C for the first and C for the second.

$$\begin{aligned} P(c'_i = A \quad &| \quad take\_course = i, c_j = C, c_k = C) = 0.20 \\ P(c'_i = B \quad &| \quad take\_course = i, c_j = C, c_k = C) = 0.25 \\ P(c'_i = C \quad &| \quad take\_course = i, c_j = C, c_k = C) = 0.40 \\ P(c'_i = E \quad &| \quad take\_course = i, c_j = C, c_k = C) = 0.15 \\ P(c'_i = none \quad &| \quad take\_course = i, c_j = C, c_k = C) = 0 \end{aligned}$$

- The grades obtained for pre-requisite courses were C for the first and E for the second.

$$\begin{aligned} P(c'_i = A \quad &| \quad take\_course = i, c_j = C, c_k = E) = 0.10 \\ P(c'_i = B \quad &| \quad take\_course = i, c_j = C, c_k = E) = 0.20 \\ P(c'_i = C \quad &| \quad take\_course = i, c_j = C, c_k = E) = 0.30 \\ P(c'_i = E \quad &| \quad take\_course = i, c_j = C, c_k = E) = 0.40 \\ P(c'_i = none \quad &| \quad take\_course = i, c_j = C, c_k = E) = 0 \end{aligned}$$

- The grades obtained for pre-requisite courses were C for the first and $none$ for the second which was not taken.

$$\begin{aligned} P(c'_i = A \quad &| \quad take\_course = i, c_j = C, c_k = none) = 0.15 \\ P(c'_i = B \quad &| \quad take\_course = i, c_j = C, c_k = none) = 0.25 \\ P(c'_i = C \quad &| \quad take\_course = i, c_j = C, c_k = none) = 0.35 \\ P(c'_i = E \quad &| \quad take\_course = i, c_j = C, c_k = none) = 0.25 \\ P(c'_i = none \quad &| \quad take\_course = i, c_j = C, c_k = none) = 0 \end{aligned}$$

One pre-requisite mark is E.

- The grades obtained for pre-requisite courses were E for the first and A for the second.

$$\begin{aligned} P(c'_i = A \quad &| \quad take\_course = i, c_j = E, c_k = A) = 0.30 \\ P(c'_i = B \quad &| \quad take\_course = i, c_j = E, c_k = A) = 0.30 \\ P(c'_i = C \quad &| \quad take\_course = i, c_j = E, c_k = A) = 0.20 \\ P(c'_i = E \quad &| \quad take\_course = i, c_j = E, c_k = A) = 0.20 \\ P(c'_i = none \quad &| \quad take\_course = i, c_j = E, c_k = A) = 0 \end{aligned}$$

- The grades obtained for pre-requisite courses were E for the first and B for the second.

$$\begin{aligned}
P(c_i' = A \quad &| \quad take_course = i, c_j = E, c_k = B) = 0.20 \\
P(c_i' = B \quad &| \quad take_course = i, c_j = E, c_k = B) = 0.25 \\
P(c_i' = C \quad &| \quad take_course = i, c_j = E, c_k = B) = 0.35 \\
P(c_i' = E \quad &| \quad take_course = i, c_j = E, c_k = B) = 0.20 \\
P(c_i' = none \quad &| \quad take_course = i, c_j = E, c_k = B) = 0
\end{aligned}$$

- The grades obtained for pre-requisite courses were E for the first and C for the second.

$$\begin{aligned}
P(c_i' = A \quad &| \quad take_course = i, c_j = E, c_k = C) = 0.10 \\
P(c_i' = B \quad &| \quad take_course = i, c_j = E, c_k = C) = 0.20 \\
P(c_i' = C \quad &| \quad take_course = i, c_j = E, c_k = C) = 0.30 \\
P(c_i' = E \quad &| \quad take_course = i, c_j = E, c_k = C) = 0.40 \\
P(c_i' = none \quad &| \quad take_course = i, c_j = E, c_k = C) = 0
\end{aligned}$$

- The grades obtained for pre-requisite courses were E for the first and E for the second.

$$\begin{aligned}
P(c_i' = A \quad &| \quad take_course = i, c_j = E, c_k = E) = 0.02 \\
P(c_i' = B \quad &| \quad take_course = i, c_j = E, c_k = E) = 0.05 \\
P(c_i' = C \quad &| \quad take_course = i, c_j = E, c_k = E) = 0.23 \\
P(c_i' = E \quad &| \quad take_course = i, c_j = E, c_k = E) = 0.70 \\
P(c_i' = none \quad &| \quad take_course = i, c_j = E, c_k = E) = 0
\end{aligned}$$

- The grades obtained for pre-requisite courses were E for the first and $none$ for the second which was not taken.

$$\begin{aligned}
P(c_i' = A \quad &| \quad take_course = i, c_j = E, c_k = none) = 0.05 \\
P(c_i' = B \quad &| \quad take_course = i, c_j = E, c_k = none) = 0.10 \\
P(c_i' = C \quad &| \quad take_course = i, c_j = E, c_k = none) = 0.30 \\
P(c_i' = E \quad &| \quad take_course = i, c_j = E, c_k = none) = 0.55 \\
P(c_i' = none \quad &| \quad take_course = i, c_j = E, c_k = none) = 0
\end{aligned}$$

One pre-requisite mark is $none$.

- The grades obtained for pre-requisite courses were $none$ for the first and A for the second.

$$\begin{aligned}
P(c'_i = A \quad &| \quad take_course = i, c_j = none, c_k = A) = 0.40 \\
P(c'_i = B \quad &| \quad take_course = i, c_j = none, c_k = A) = 0.30 \\
P(c'_i = C \quad &| \quad take_course = i, c_j = none, c_k = A) = 0.20 \\
P(c'_i = E \quad &| \quad take_course = i, c_j = none, c_k = A) = 0.10 \\
P(c'_i = none \quad &| \quad take_course = i, c_j = none, c_k = A) = 0
\end{aligned}$$

- The grades obtained for pre-requisite courses were $none$ for the first and B for the second.

$$\begin{aligned}
P(c'_i = A \quad &| \quad take_course = i, c_j = none, c_k = B) = 0.30 \\
P(c'_i = B \quad &| \quad take_course = i, c_j = none, c_k = B) = 0.35 \\
P(c'_i = C \quad &| \quad take_course = i, c_j = none, c_k = B) = 0.25 \\
P(c'_i = E \quad &| \quad take_course = i, c_j = none, c_k = B) = 0.10 \\
P(c'_i = none \quad &| \quad take_course = i, c_j = none, c_k = B) = 0
\end{aligned}$$

- The grades obtained for pre-requisite courses were $none$ for the first and C for the second.

$$\begin{aligned}
P(c'_i = A \quad &| \quad take_course = i, c_j = none, c_k = C) = 0.15 \\
P(c'_i = B \quad &| \quad take_course = i, c_j = none, c_k = C) = 0.25 \\
P(c'_i = C \quad &| \quad take_course = i, c_j = none, c_k = C) = 0.35 \\
P(c'_i = E \quad &| \quad take_course = i, c_j = none, c_k = C) = 0.25 \\
P(c'_i = none \quad &| \quad take_course = i, c_j = none, c_k = C) = 0
\end{aligned}$$

- The grades obtained for pre-requisite courses were $none$ for the first and E for the

second.

$$\begin{aligned}
P(c_i' = A \quad &| \quad take_course = i, c_j = none, c_k = E) = 0.05 \\
P(c_i' = B \quad &| \quad take_course = i, c_j = none, c_k = E) = 0.10 \\
P(c_i' = C \quad &| \quad take_course = i, c_j = none, c_k = E) = 0.30 \\
P(c_i' = E \quad &| \quad take_course = i, c_j = none, c_k = E) = 0.55 \\
P(c_i' = none \quad &| \quad take_course = i, c_j = none, c_k = E) = 0
\end{aligned}$$

- The grades obtained for pre-requisite courses were $none$ for the first and $none$ for the second which was not taken.

$$\begin{aligned}
P(c_i' = A \quad &| \quad take_course = i, c_j = none, c_k = none) = 0.25 \\
P(c_i' = B \quad &| \quad take_course = i, c_j = none, c_k = none) = 0.30 \\
P(c_i' = C \quad &| \quad take_course = i, c_j = none, c_k = none) = 0.30 \\
P(c_i' = E \quad &| \quad take_course = i, c_j = none, c_k = none) = 0.15 \\
P(c_i' = none \quad &| \quad take_course = i, c_j = none, c_k = none) = 0
\end{aligned}$$

Where $i = 1, \ldots, n$, and c_j and c_k are the pre-requisite course variables.

Rewards The reward model is quite simply the sum of mean number of credits for each course taken.

$$R(\boldsymbol{c}) = \sum_{i=1}^{n} \sum_{m} P(c_i = m) \text{ reward}(c_i)$$

Where

$$\text{reward}(c_i) = \begin{cases} \text{num\_credits}(c_i) & \text{if } c_i \in \{A, B, C\} \\ 0 & \text{otherwise} \end{cases}$$

and the number of credits num_credits(c_i) is shown in the table below.

Course	Number of credits
c_1	3
c_2	3
c_3	2
c_4	4
c_5	4
c_6	3
c_7	3
c_8	3
c_9	2
c_{10}	4

Appendix B

Complete Results for Chapter 5 Experiments

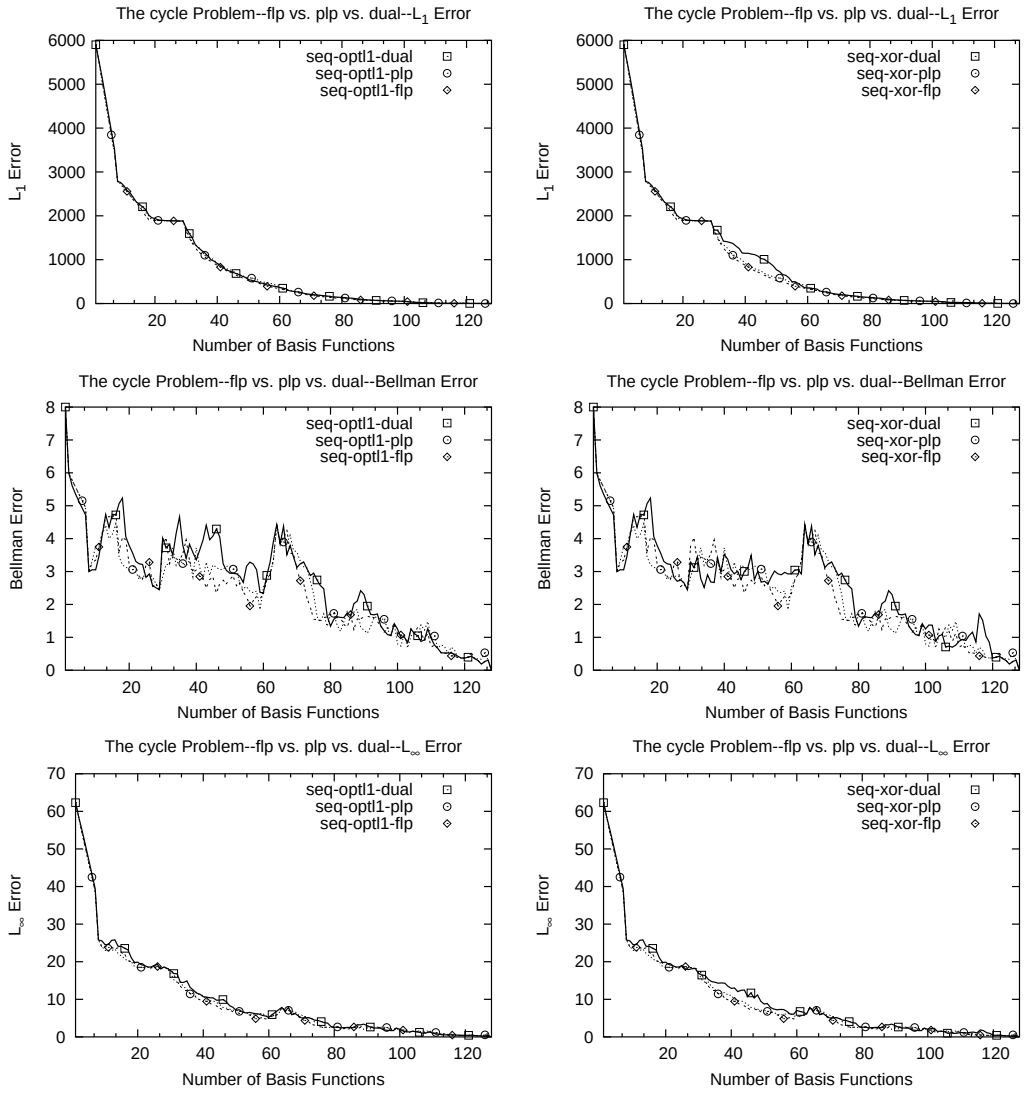


Figure B.1: Comparison of the full LP (flp), partial LP (plp) and dual scoring methods on the `cycle` problem, with basis function domains generated in a *sequential* fashion (seq), with optimized (opt) basis functions on the left, and XOR (xor) basis function.

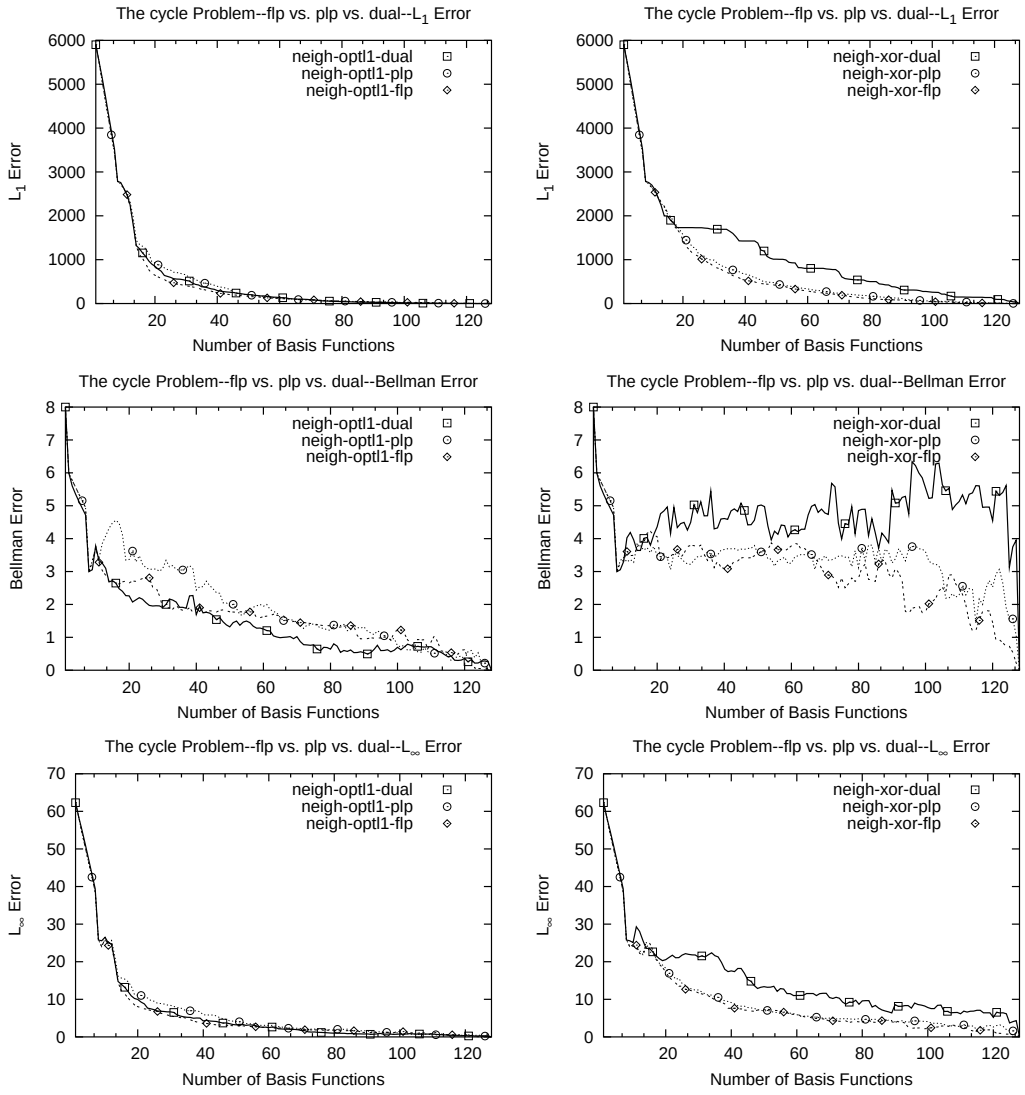


Figure B.2: Comparison of the full LP (flp), partial LP (plp) and dual scoring methods on the `cycle` problem, with basis function domains generated in a *neighbor* fashion (neigh), with optimized (opt) basis functions on the left, and XOR (xor) basis function.

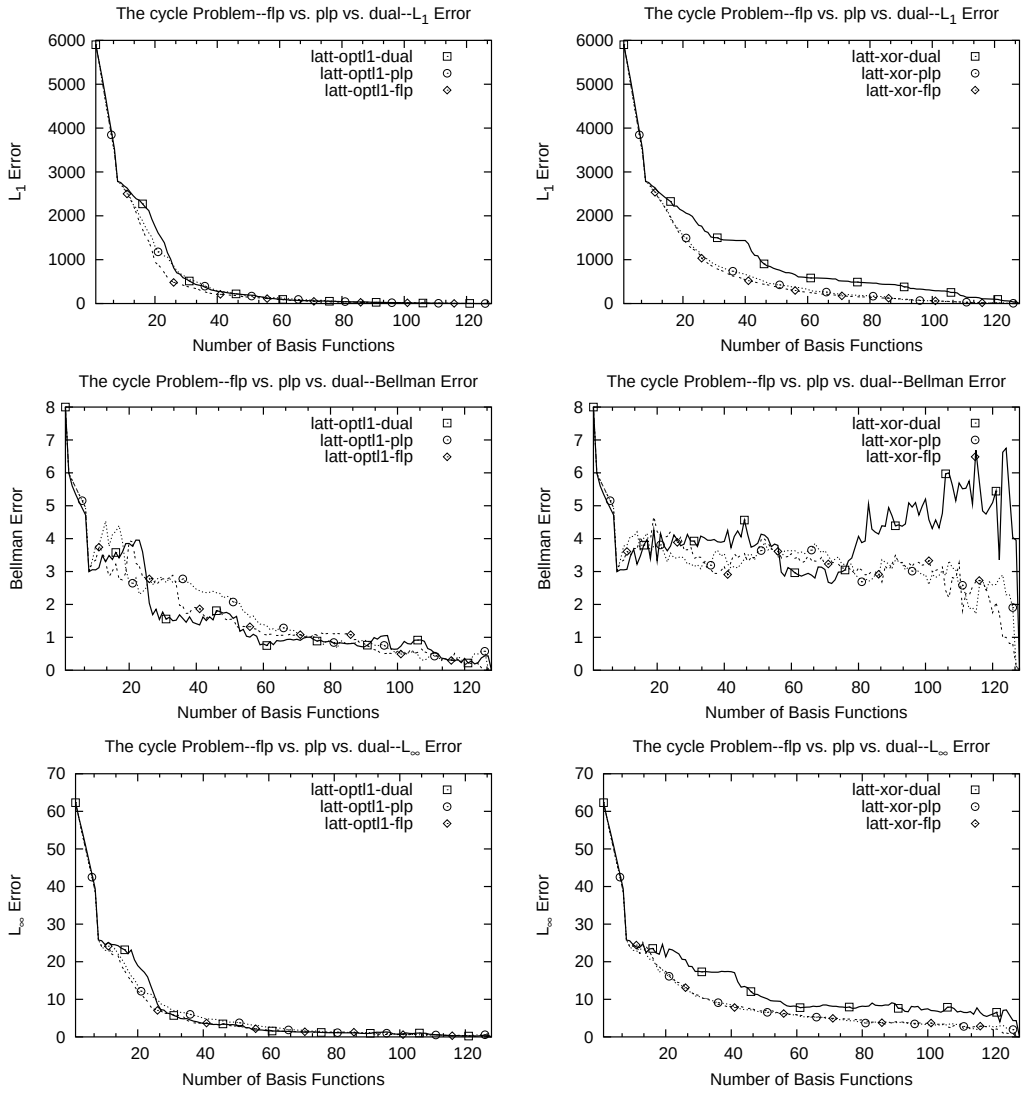


Figure B.3: Comparison of the full LP (flp), partial LP (plp) and dual scoring methods on the cycle problem, with basis function domains generated in a *lattice* fashion (latt), with optimized (opt) basis functions on the left, and XOR (xor) basis function.

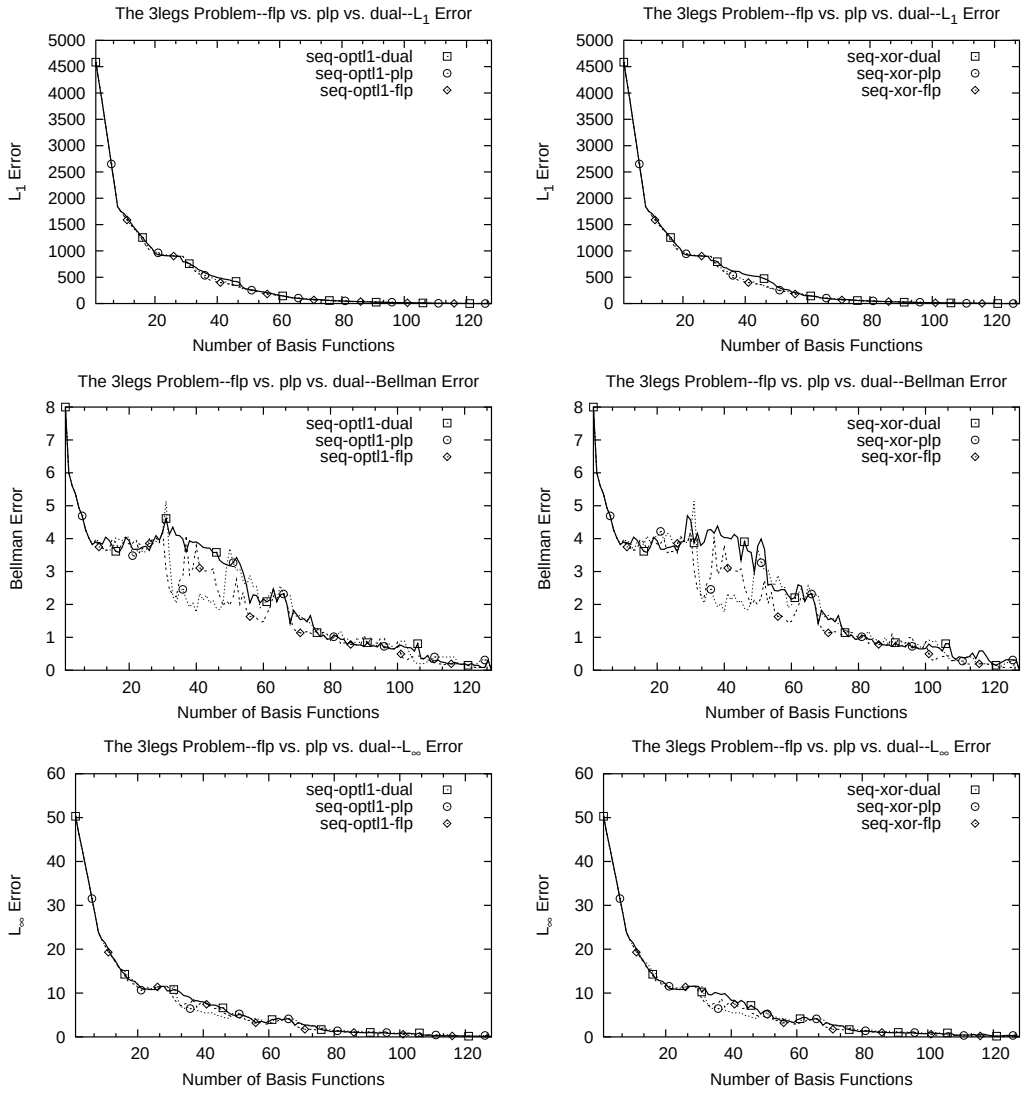


Figure B.4: Comparison of the full LP (flp), partial LP (plp) and dual scoring methods on the 3legs problem, with basis function domains generated in a *sequential* fashion (seq), with optimized (opt) basis functions on the left, and XOR (xor) basis function.

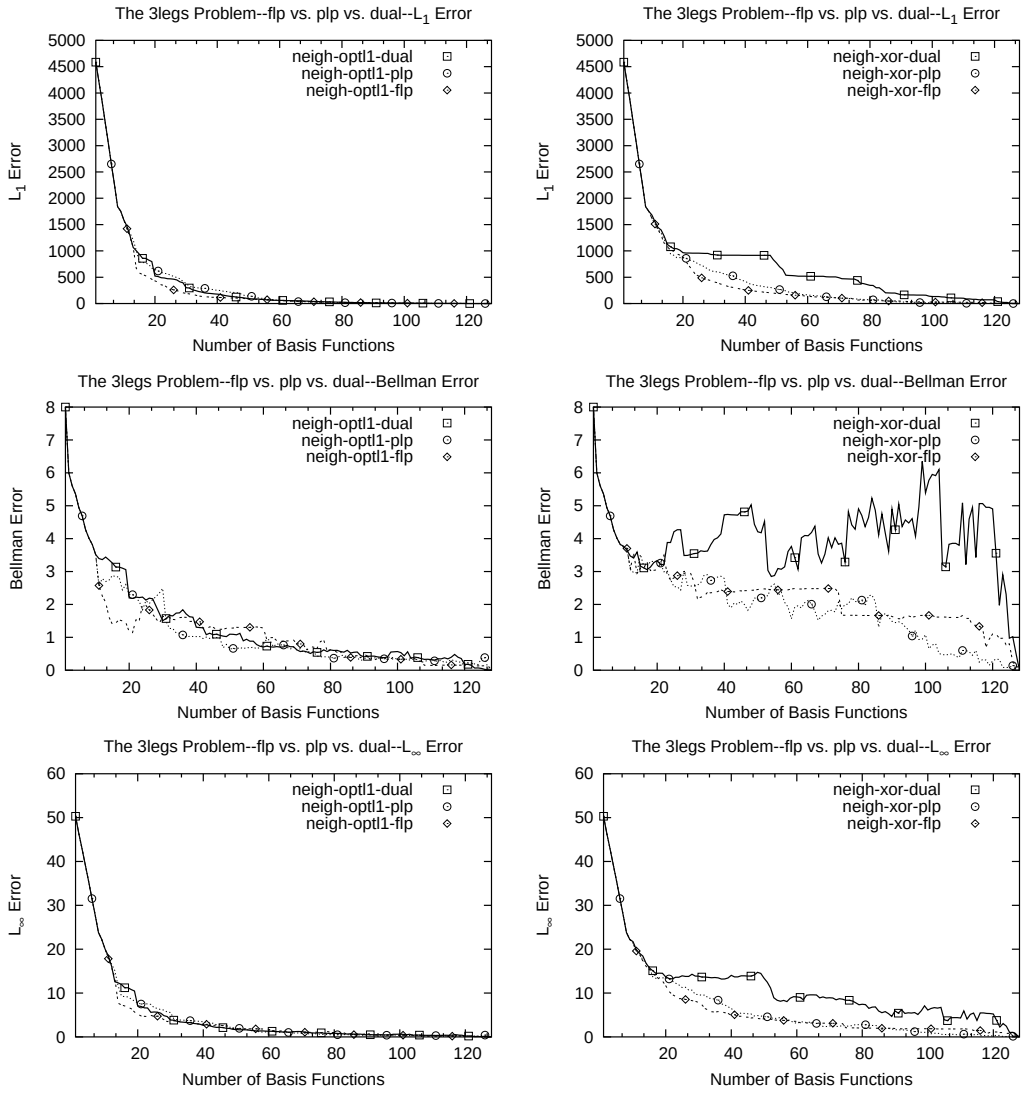


Figure B.5: Comparison of the full LP (flp), partial LP (plp) and dual scoring methods on the 3legs problem, with basis function domains generated in a *neighbor* fashion (neigh), with optimized (opt) basis functions on the left, and XOR (xor) basis function.

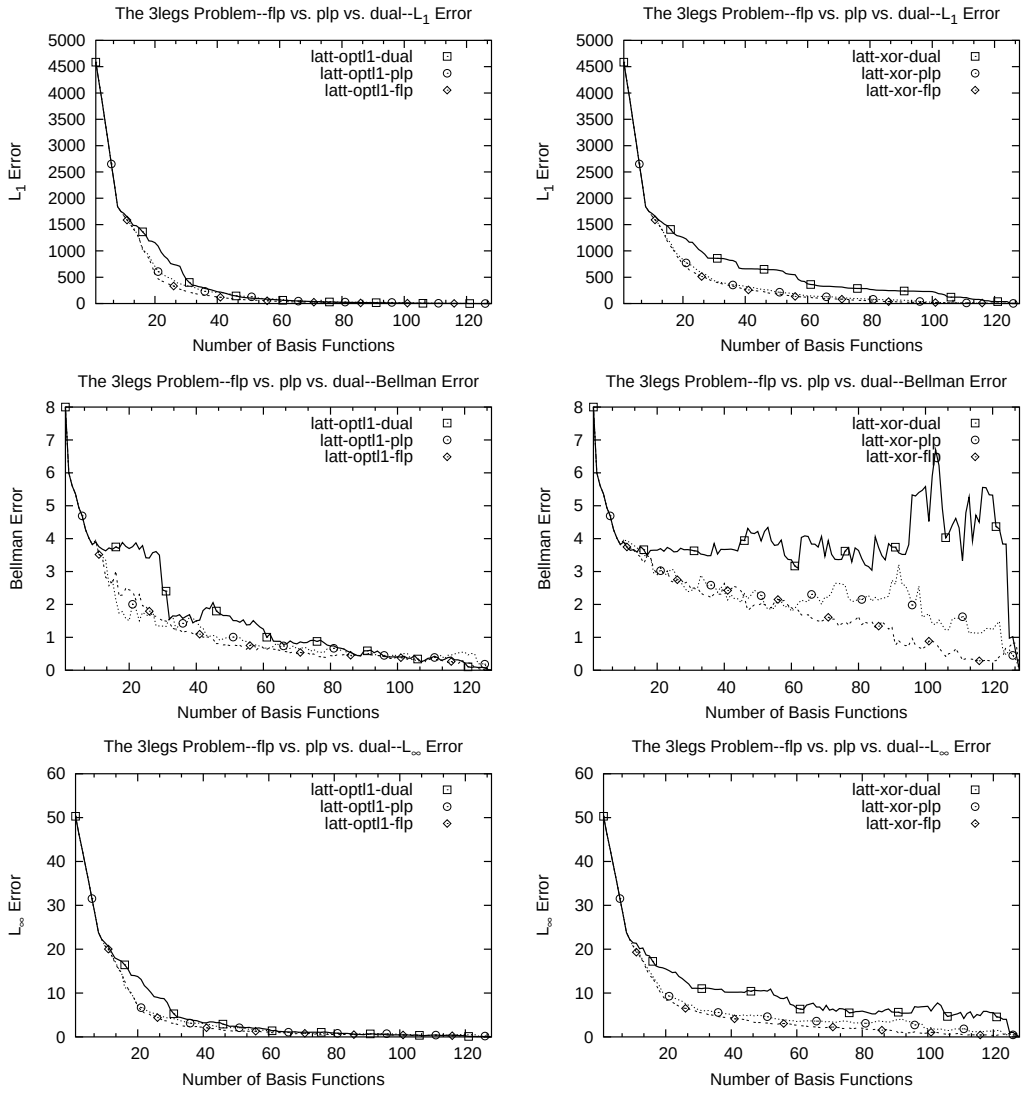


Figure B.6: Comparison of the full LP (flp), partial LP (plp) and dual scoring methods on the 3legs problem, with basis function domains generated in a *lattice* fashion (latt), with optimized (opt) basis functions on the left, and XOR (xor) basis function.

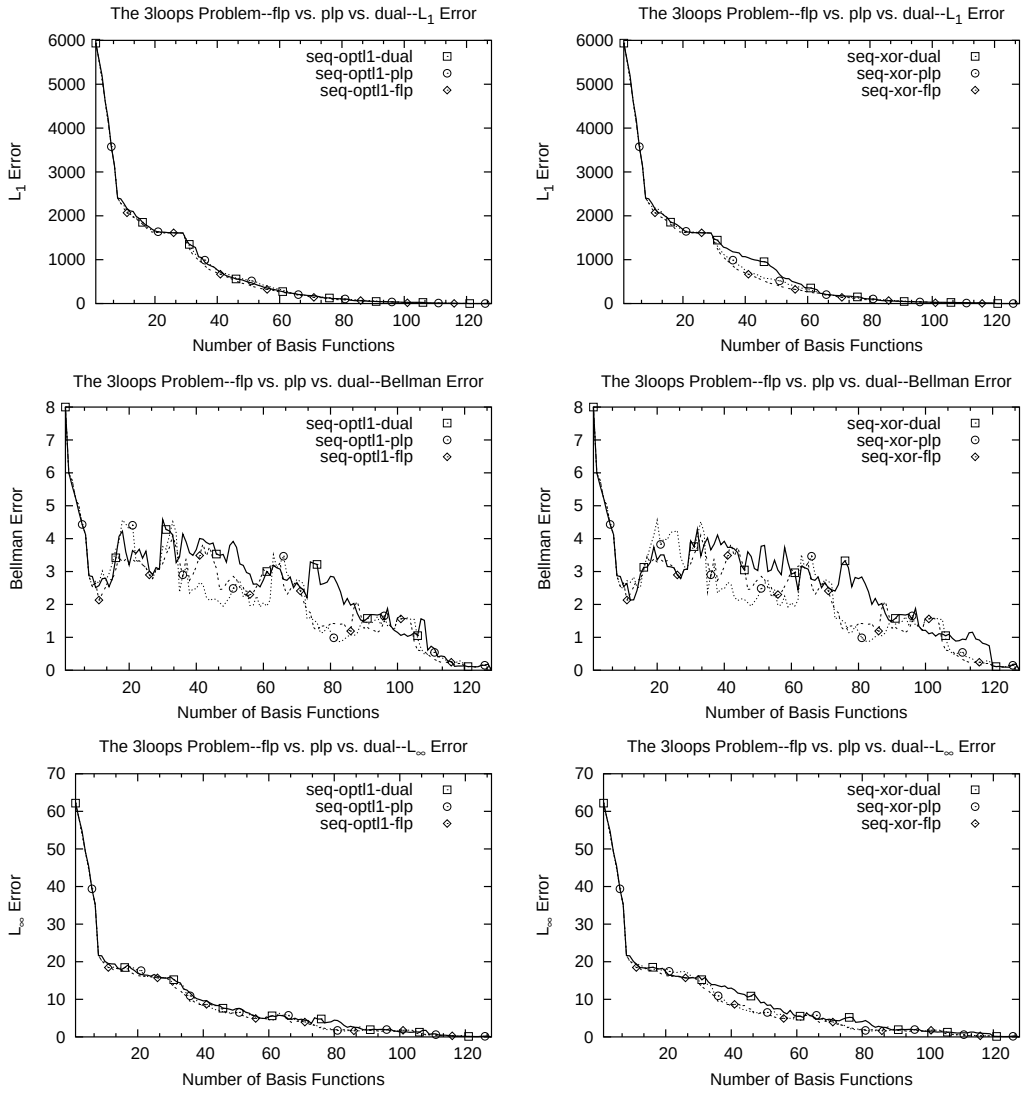


Figure B.7: Comparison of the full LP (flp), partial LP (plp) and dual scoring methods on the three loops problem, with basis function domains generated in a *sequential* fashion (seq), with optimized (opt) basis functions on the left, and XOR (xor) basis function.

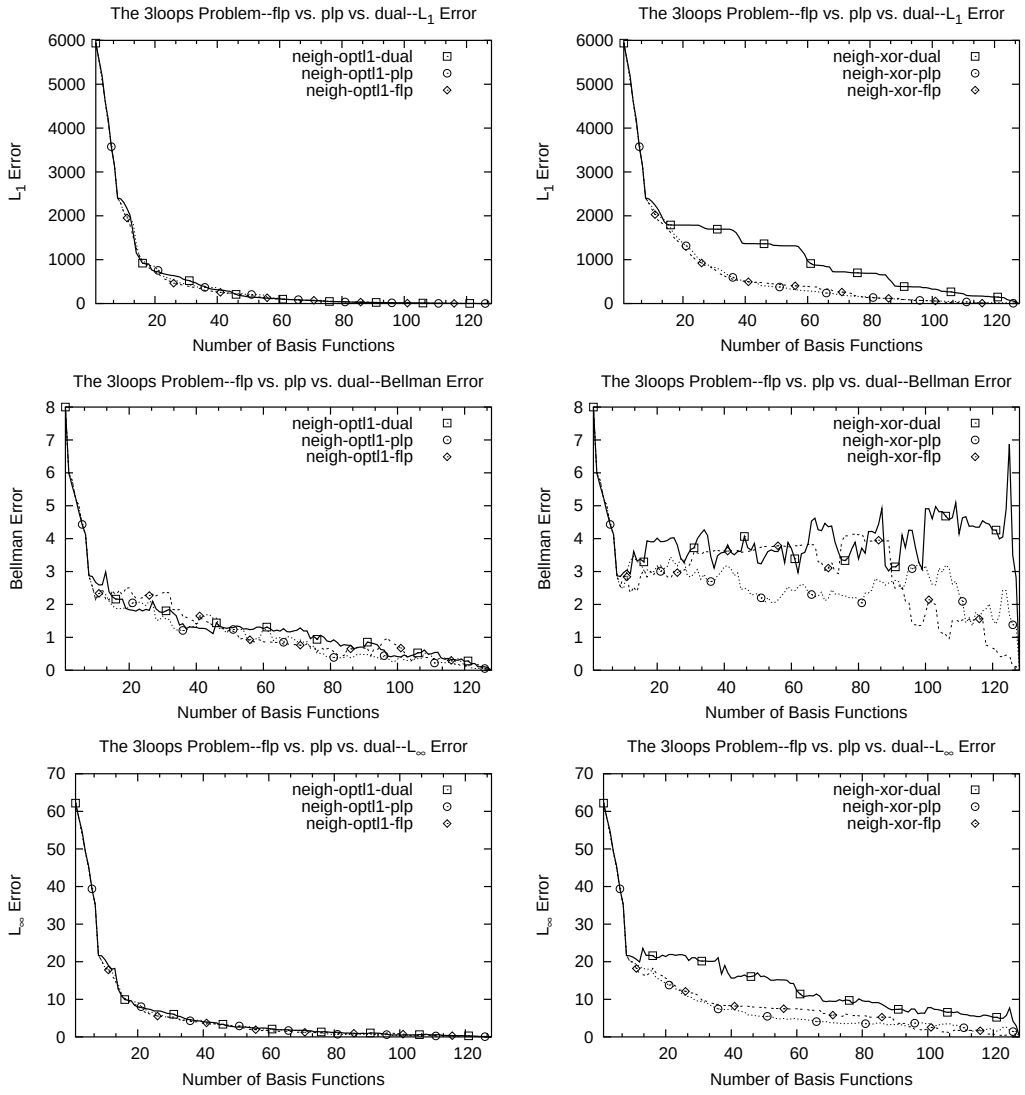


Figure B.8: Comparison of the full LP (flp), partial LP (plp) and dual scoring methods on the three loops problem, with basis function domains generated in a *neighbor* fashion (neigh), with optimized (opt) basis functions on the left, and XOR (xor) basis function.

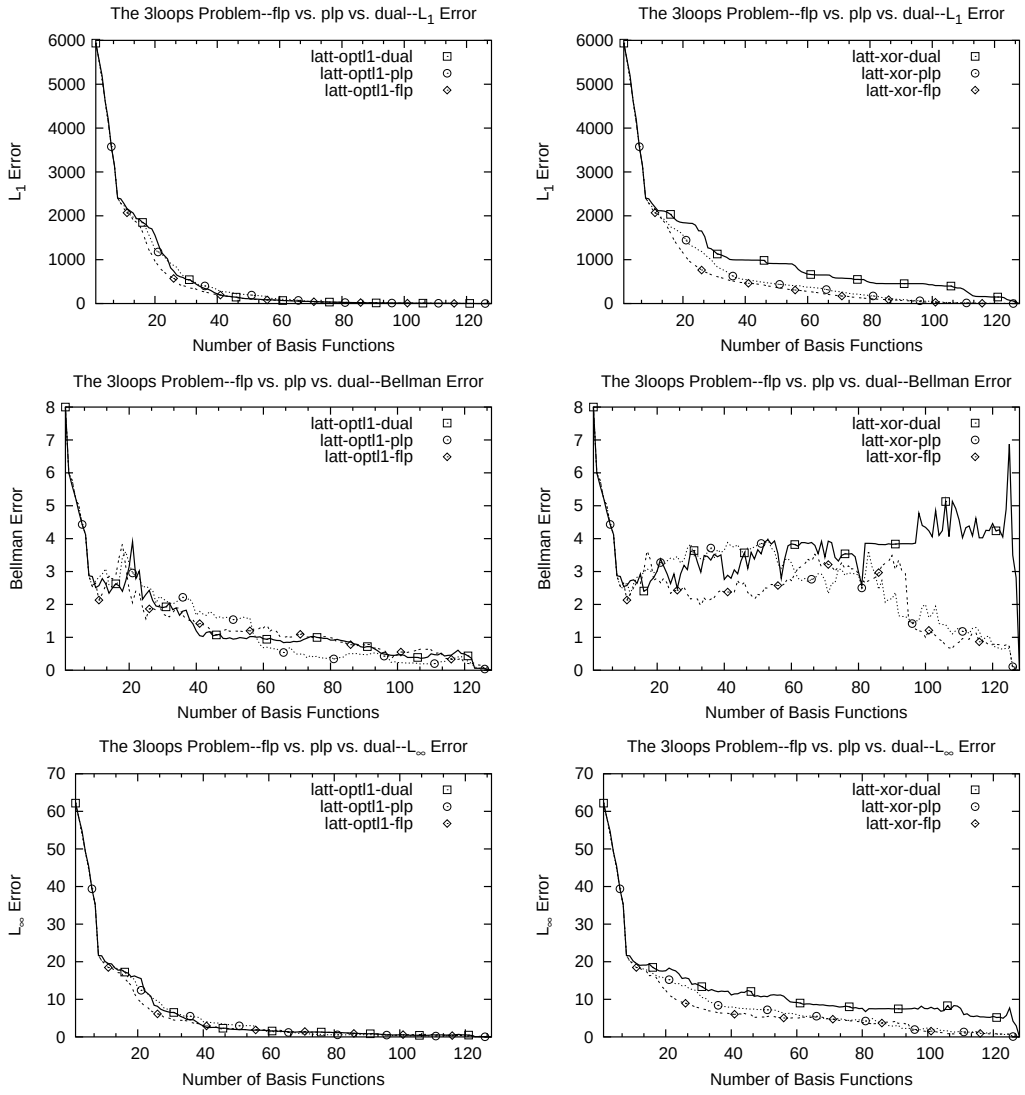


Figure B.9: Comparison of the full LP (flp), partial LP (plp) and dual scoring methods on the three loops problem, with basis function domains generated in a *lattice* fashion (latt), with optimized (opt) basis functions on the left, and XOR (xor) basis function.

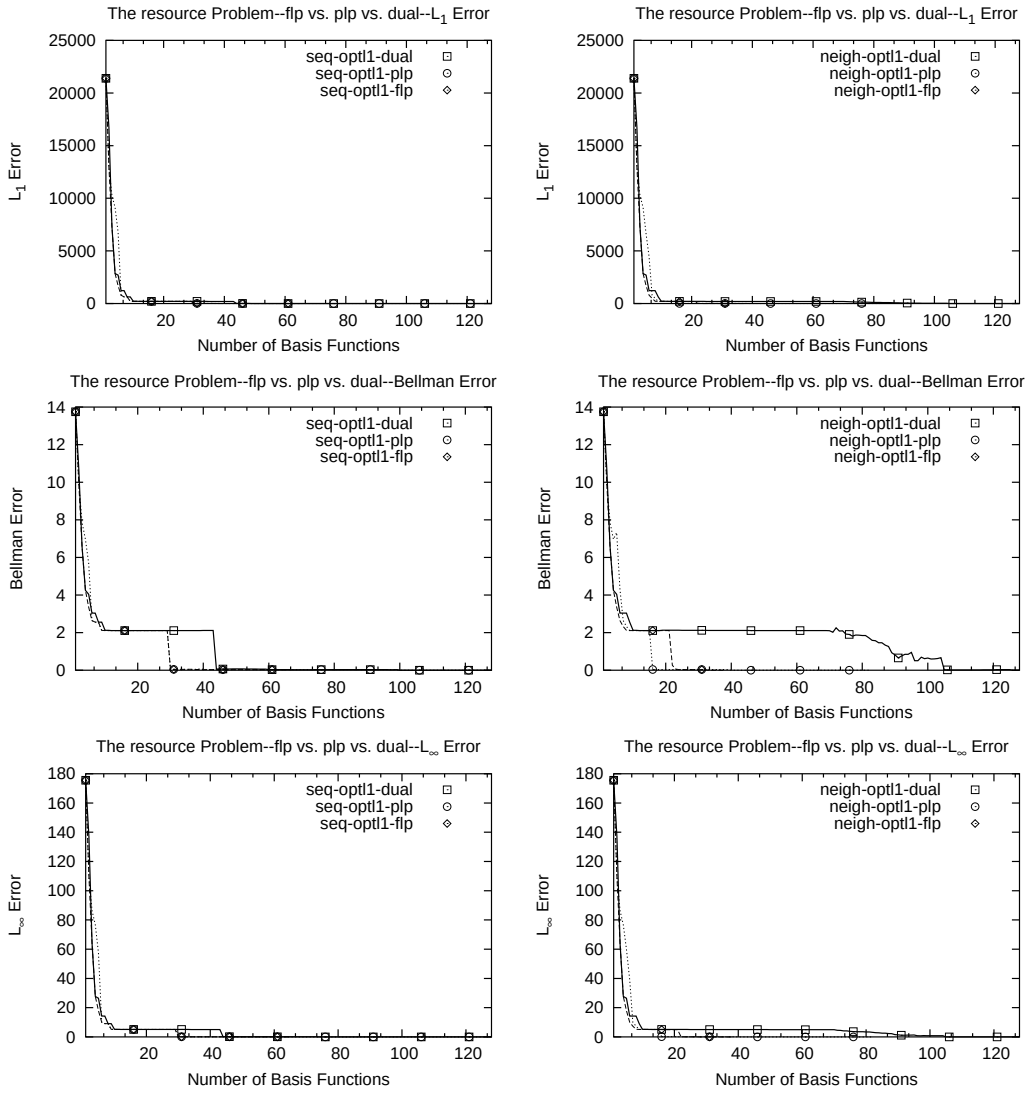


Figure B.10: Comparison of the full LP (flp), partial LP (plp) and dual scoring methods on the `resource` problem, with basis function domains generated in a *sequential* fashion (seq) on the left and *neighbor* (neigh) on the right, with optimized (opt) basis functions.

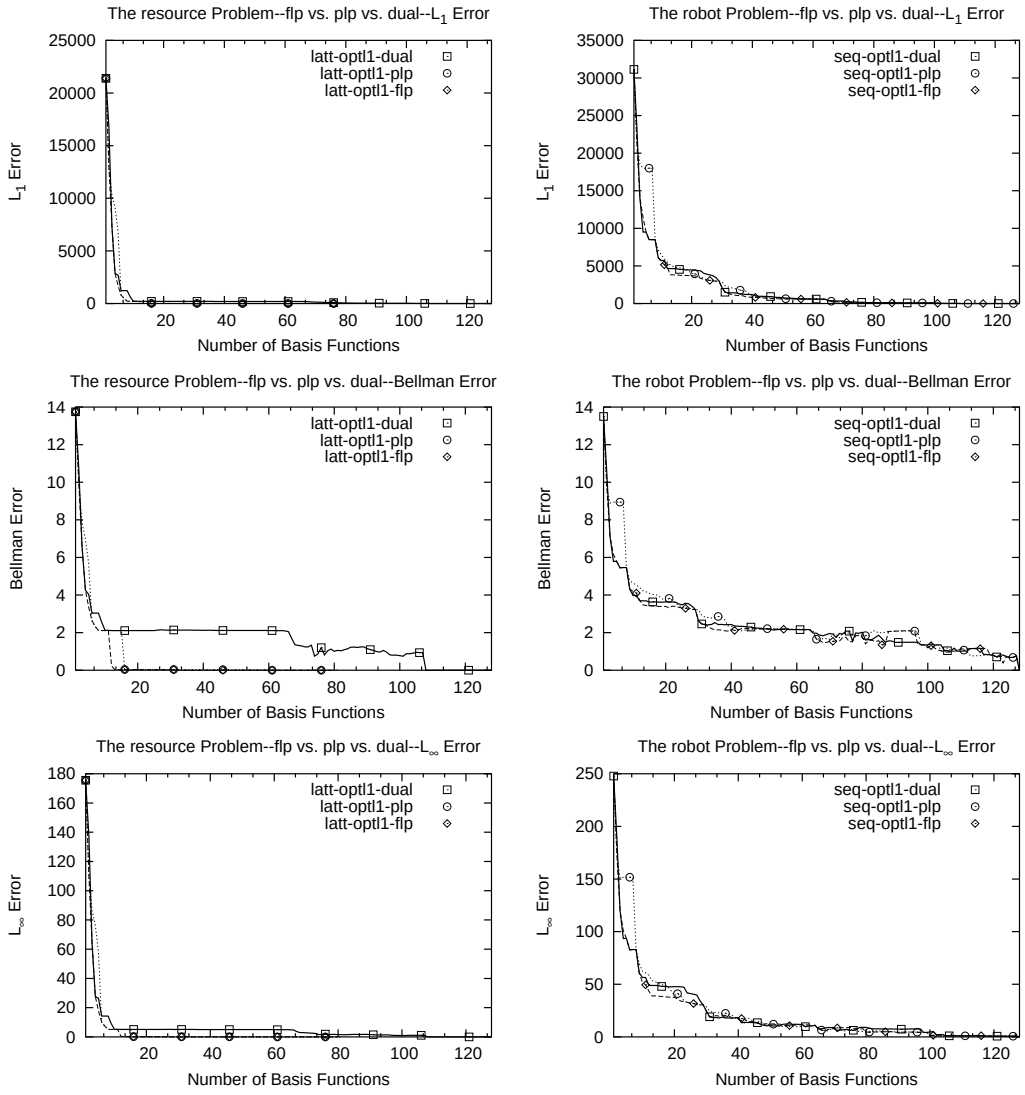


Figure B.11: Comparison of the full LP (flp), partial LP (plp) and dual scoring methods on the `resource` problem on the left and the `robot` problem on the right, with basis function domains generated in a *sequential* fashion (seq), and with optimized (opt) basis functions.

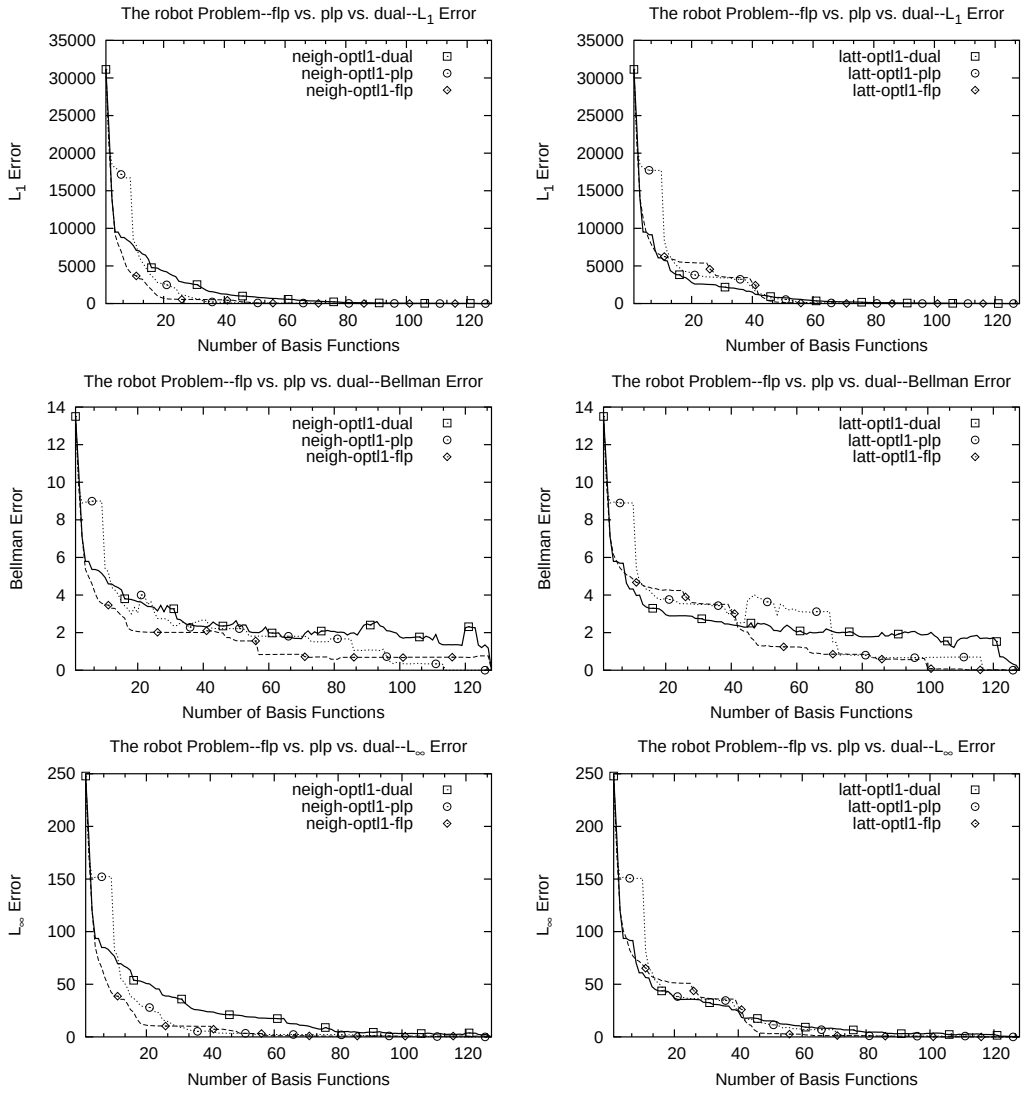


Figure B.12: Comparison of the full LP (flp), partial LP (plp) and dual scoring methods on the `robot` problem, with basis function domains generated in a *neighbor* fashion (neigh) on the left and *lattice* (latt) on the right, and with optimized (opt) basis functions.

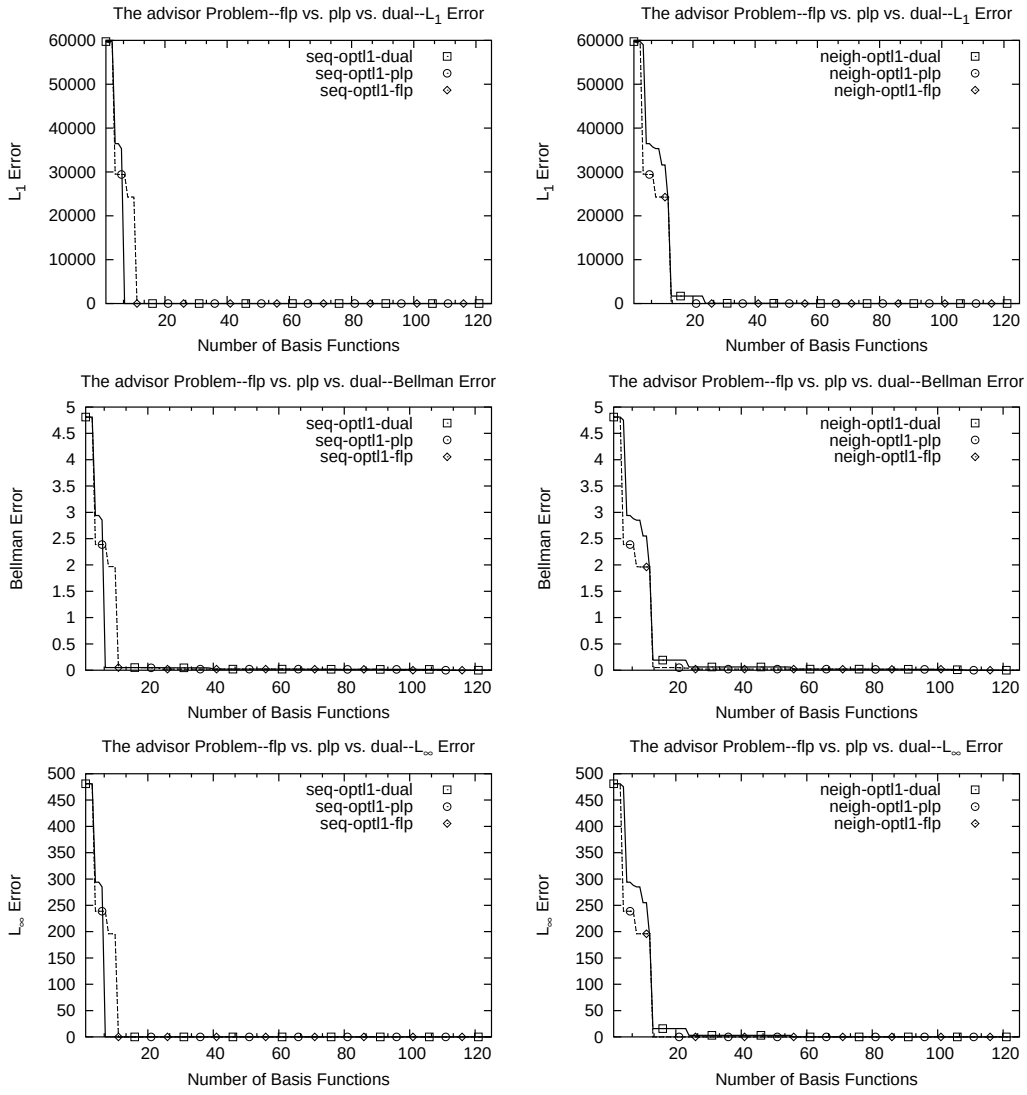


Figure B.13: Comparison of the full LP (flp), partial LP (plp) and dual scoring methods on the `advisor` problem, with basis function domains generated in a *sequential* fashion (seq) on the left and *neighbor* (neigh) on the right, and with optimized (opt) basis functions.

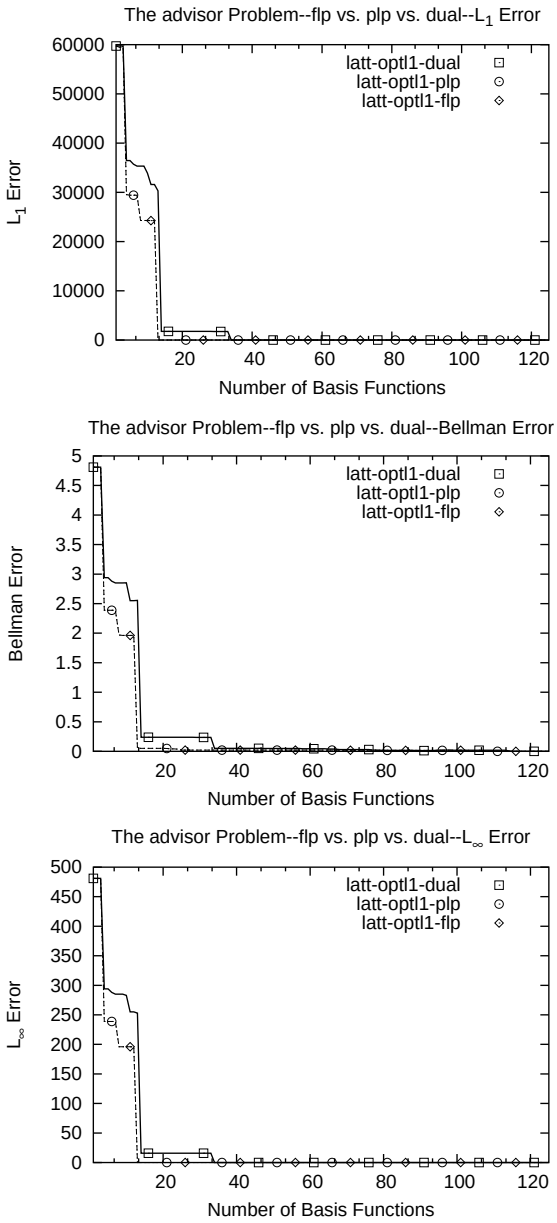


Figure B.14: Comparison of the full LP (flp), partial LP (plp) and dual scoring methods on the `advisor` problem, with basis function domains generated in a *lattice* fashion (latt), and with optimized (opt) basis functions.

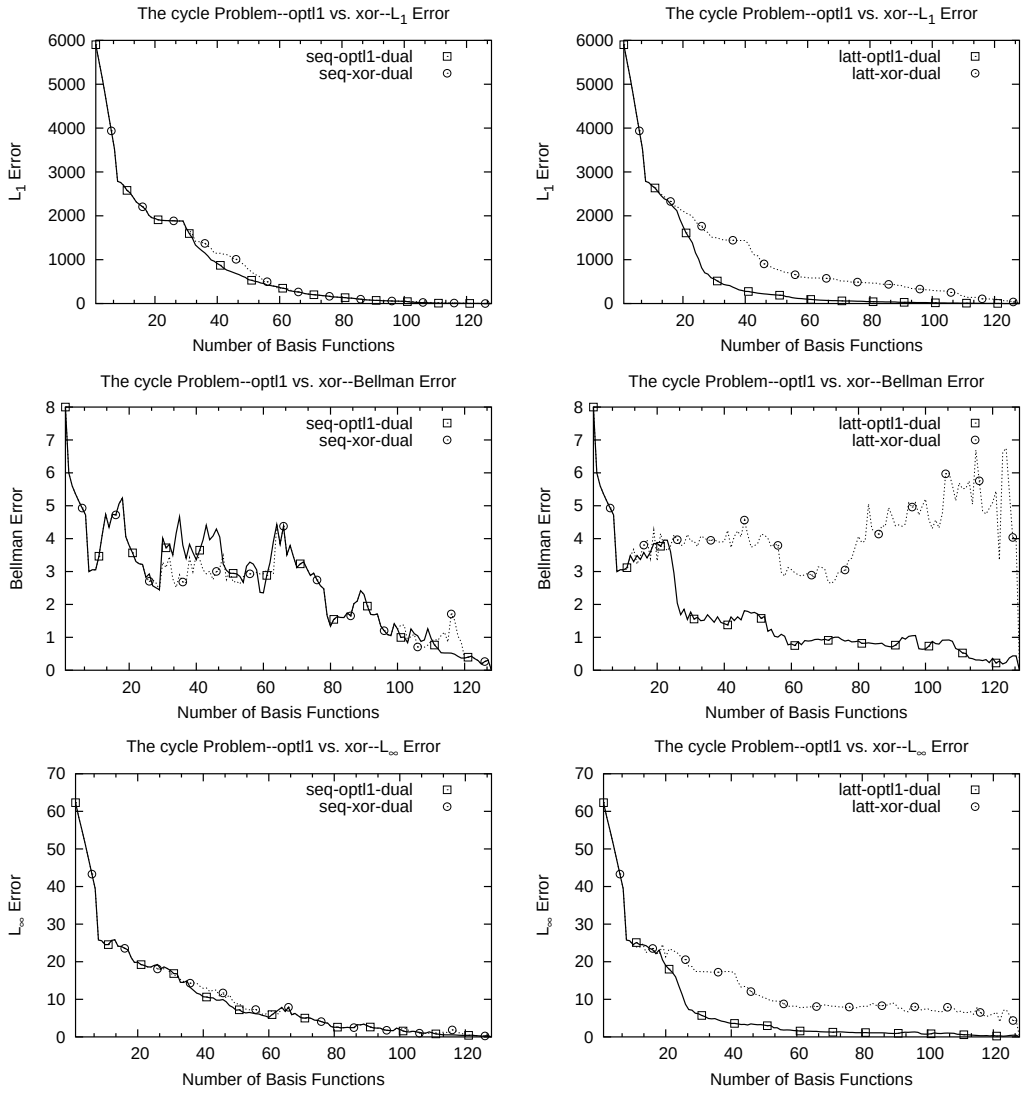


Figure B.15: Comparison of two basis function creation methods: optimized with L_1 normalization (optl1) and XOR (xor), on the `cycle` problem, with basis function domains generated in a *sequential* fashion (seq) on the left and *lattice* (latt) on the right.

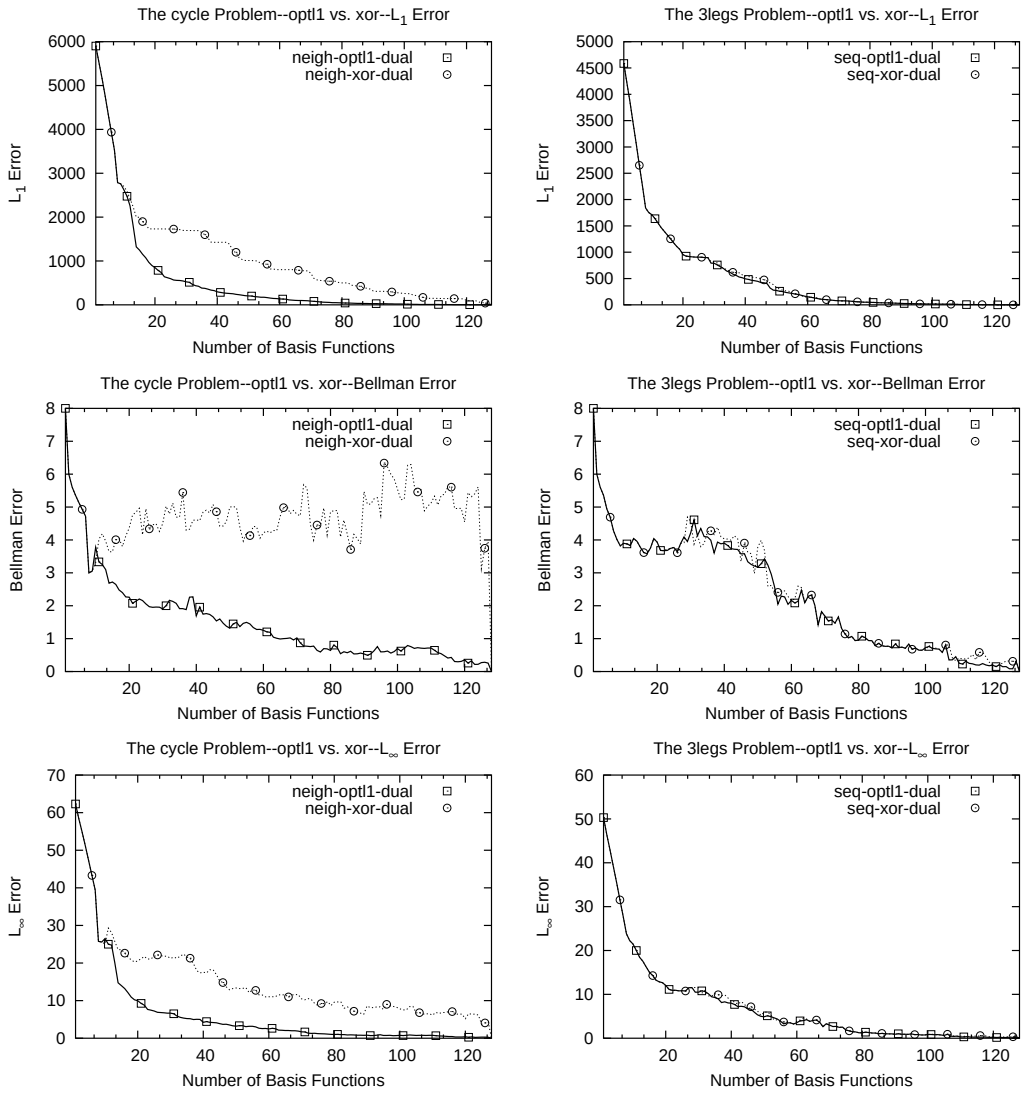


Figure B.16: Comparison of two basis function creation methods: optimized with L_1 normalization (optl1) and XOR (xor), on the `cycle` problem with basis function domains generated in a *neighbor* fashion (neigh) on the left, and `3legs` problem with basis function domains generated in a *sequential* fashion (seq) on the right.

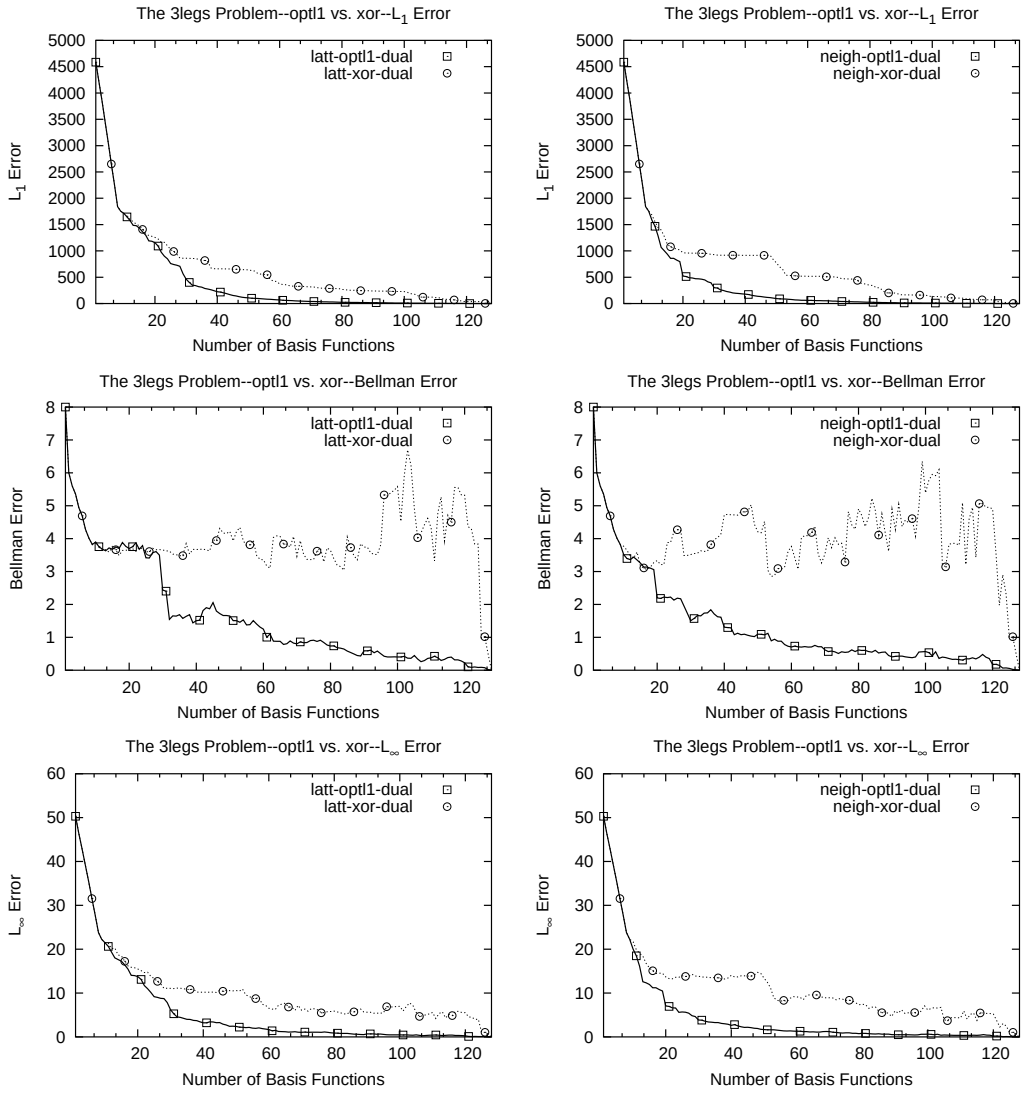


Figure B.17: Comparison of two basis function creation methods: optimized with L_1 normalization (optl1) and XOR (xor), on the `3legs` problem with basis function domains generated in a *lattice* fashion (latt) on the left and *neighbor* (neigh) on the right.

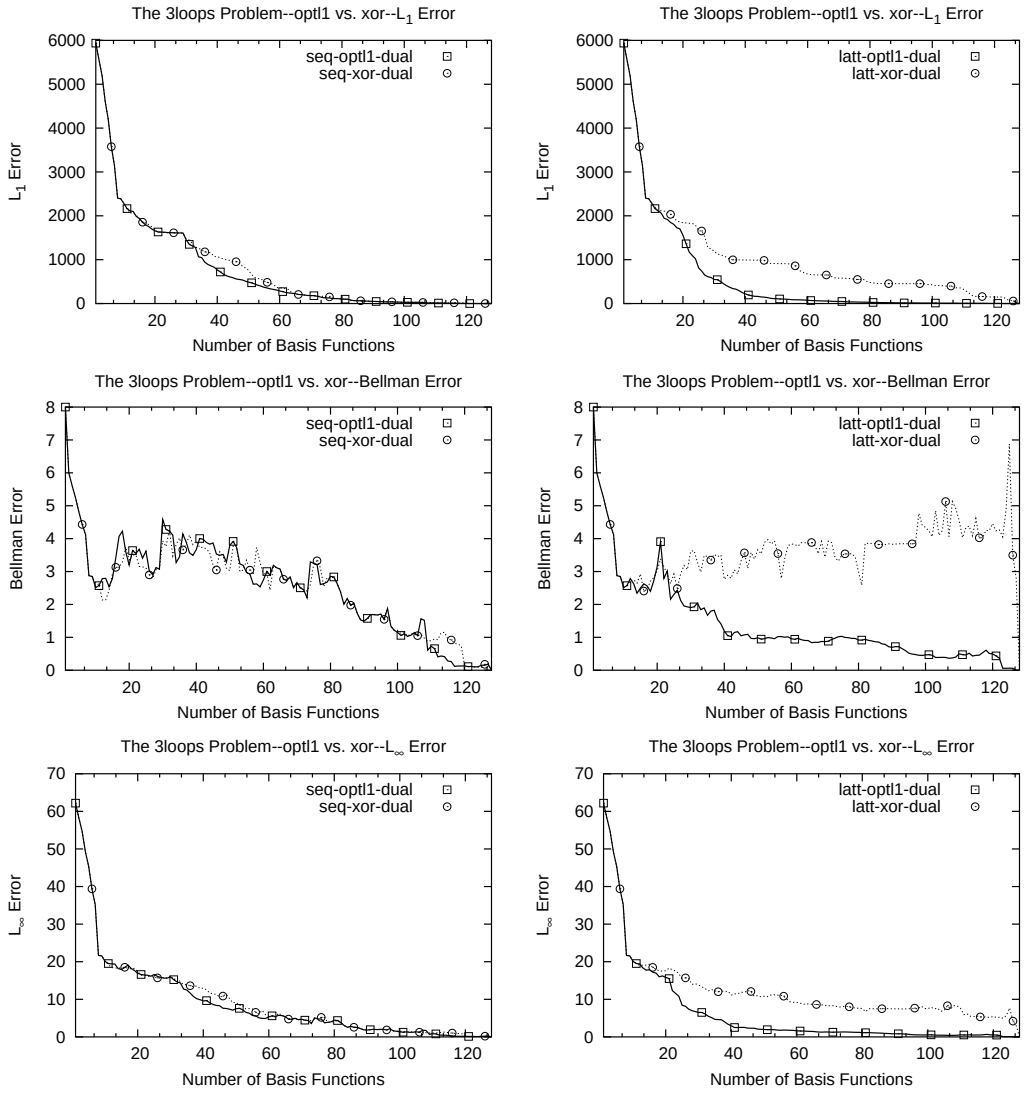


Figure B.18: Comparison of two basis function creation methods: optimized with L_1 normalization (optl1) and XOR (xor), on the three loops problem with basis function domains generated in a *sequential* fashion (seq) on the left and *lattice* (latt) on the right.

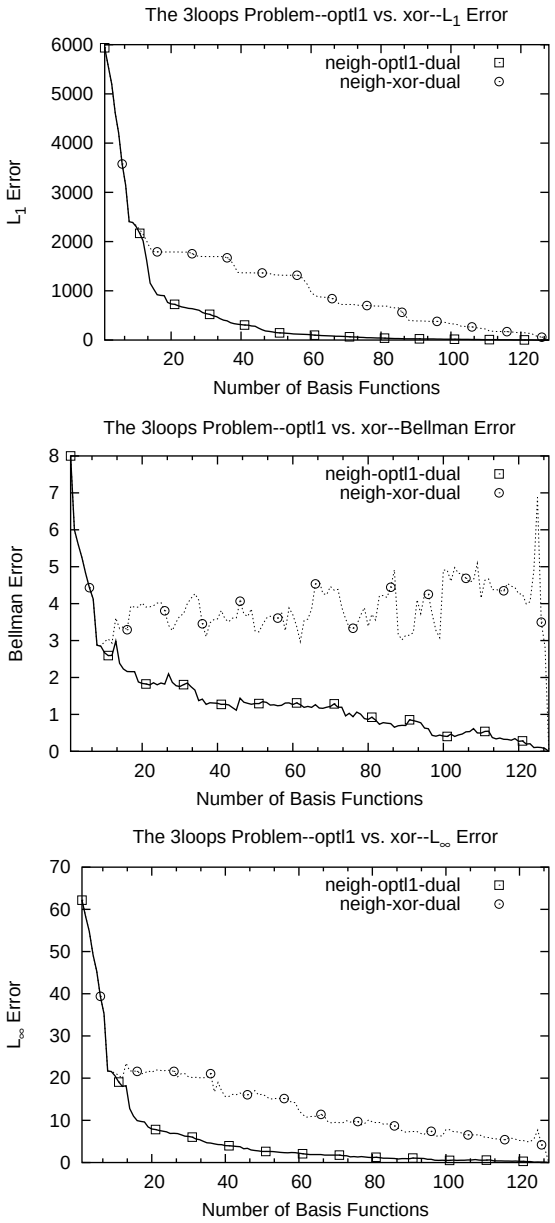


Figure B.19: Comparison of two basis function creation methods: optimized with L_1 normalization (optl1) and XOR (xor), on the three loops problem with basis function domains generated in a *neighbor* fashion (neigh).

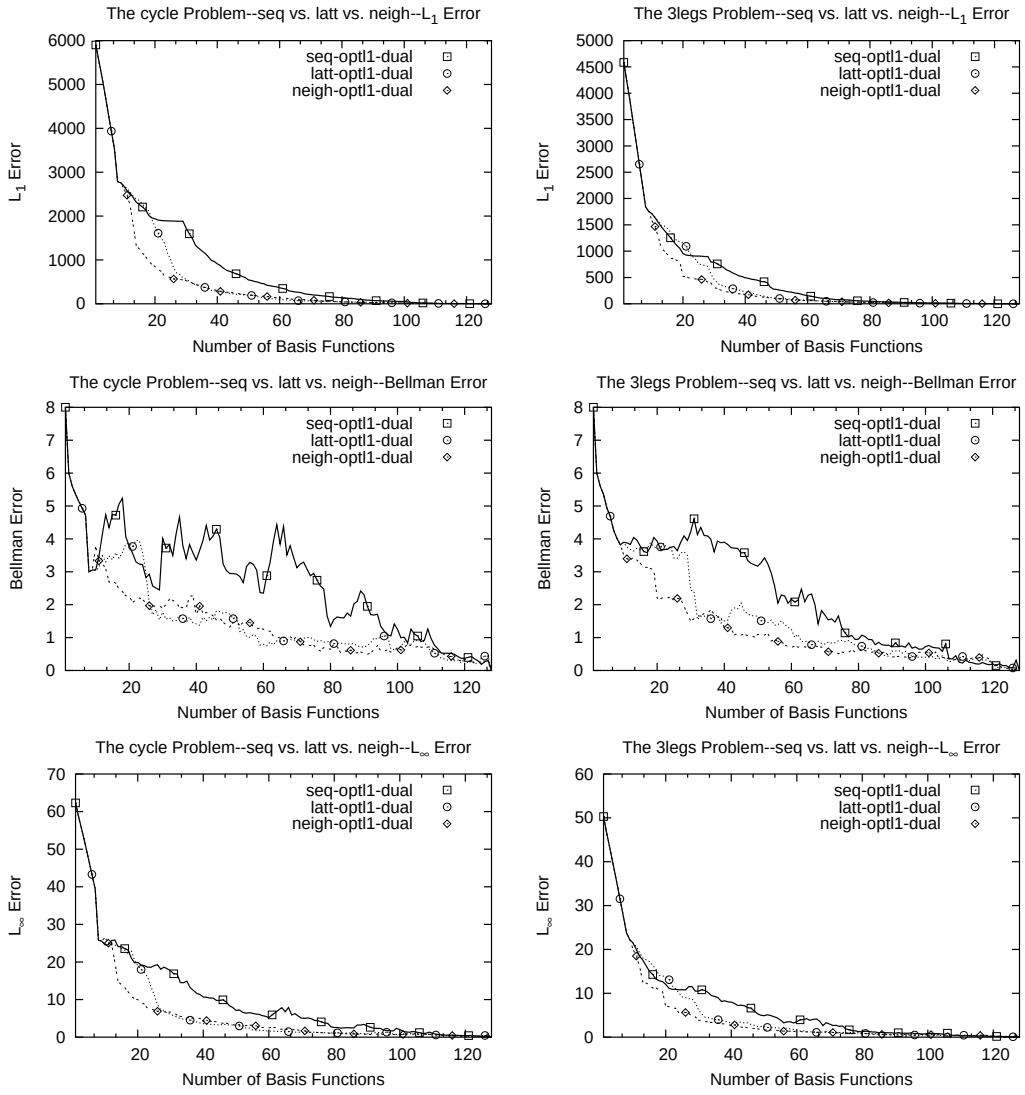


Figure B.20: Comparison of domain choosing methods `sequential` (seq), `lattice` (latt), and `neighbor` (neigh) on the `cycle` problem on the left and the `3legs` problem on the right, using optimized basis function construction with L_1 normalization (optl1) and dual score.

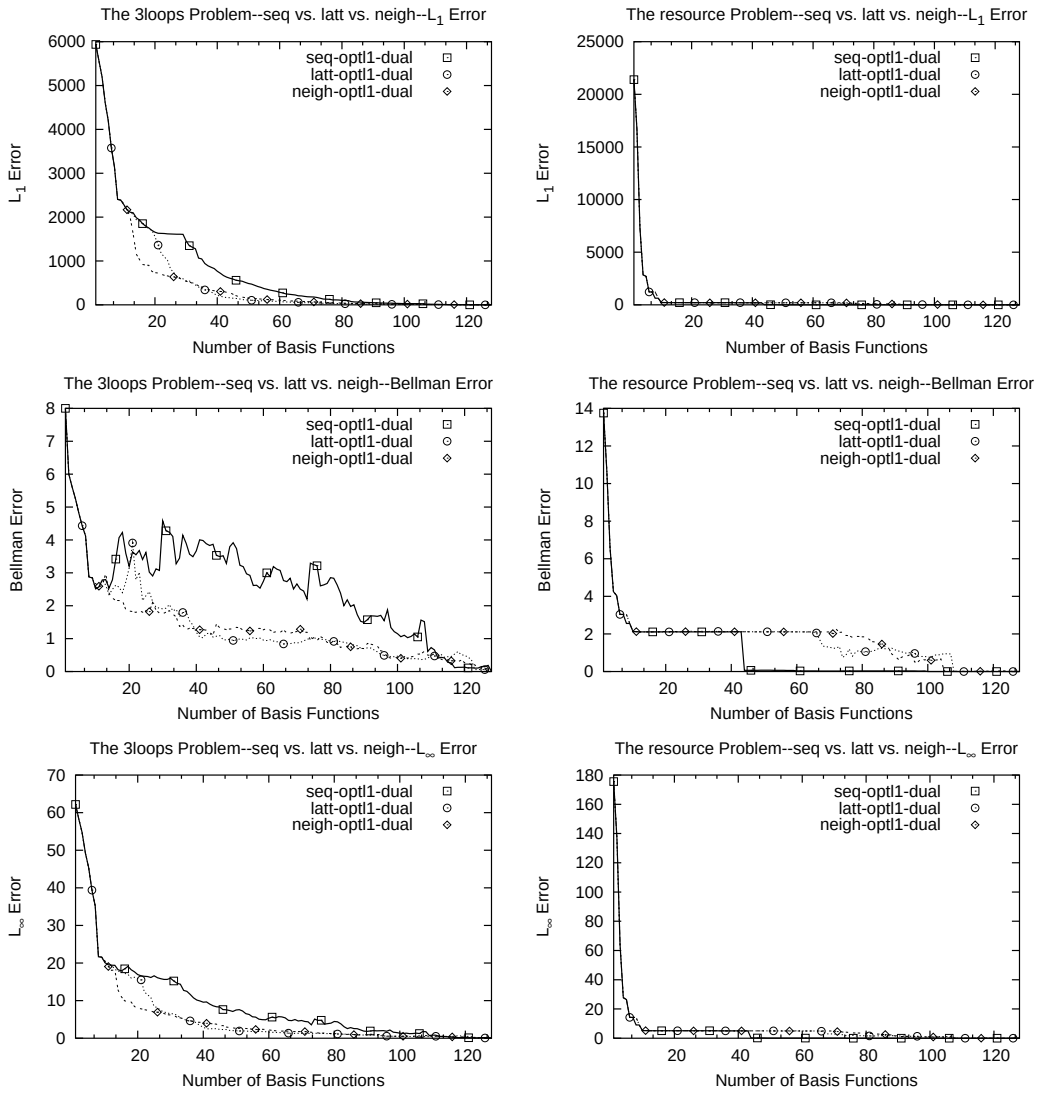


Figure B.21: Comparison of domain choosing methods `sequential` (seq), `lattice` (latt), and `neighbor` (neigh) on the `three loops` problem on the left and the `resource` problem on the right, using optimized basis function construction with L_1 normalization (optl1) and dual score.

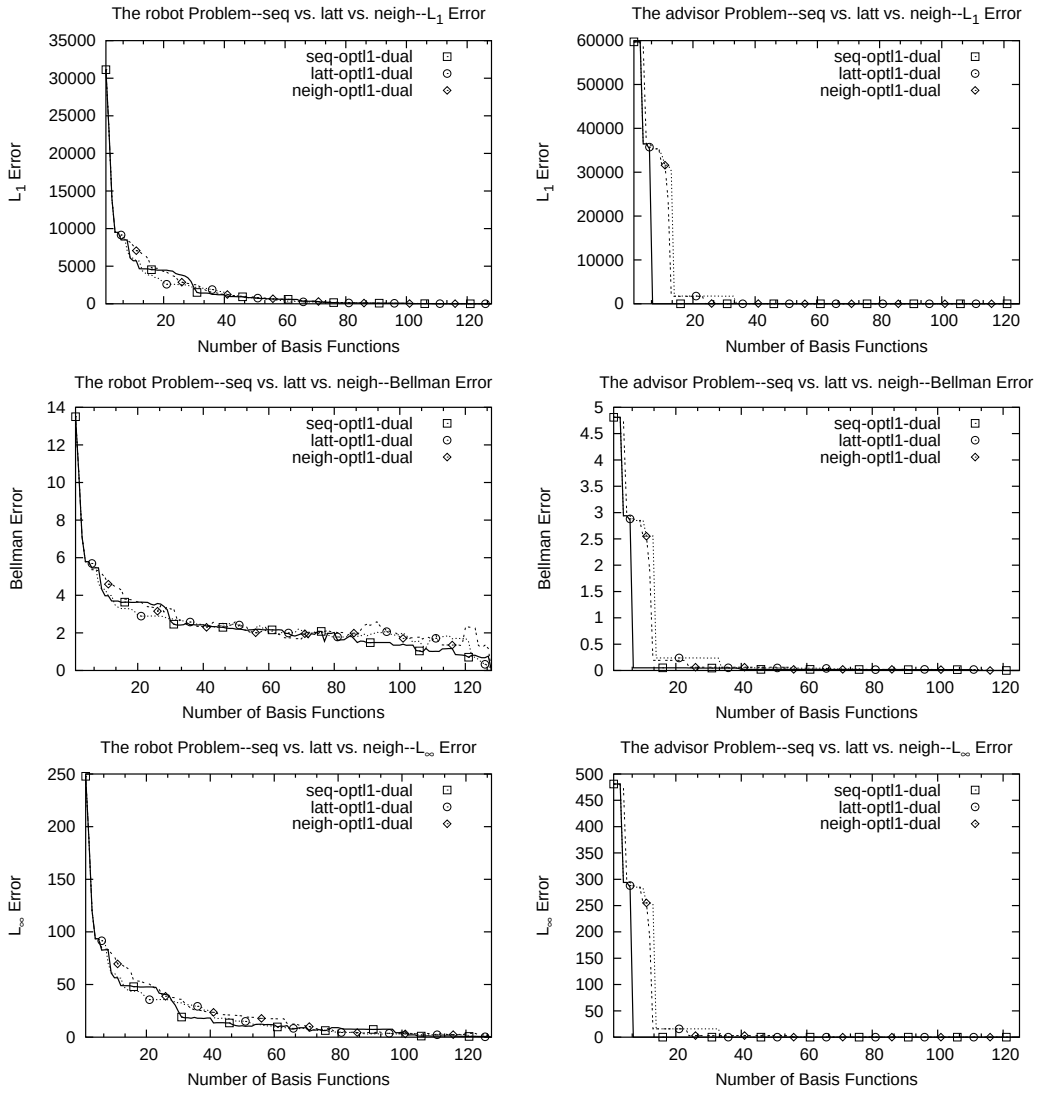


Figure B.22: Comparison of domain choosing methods `sequential` (seq), `lattice` (latt), and `neighbor` (neigh) on the `robot` problem on the left and the `advisor` problem on the right, using optimized basis function construction with L_1 normalization (optl1) and dual score.

Bibliography

[1] Fahiem Bacchus and Adam Grove. Graphical Models for Preference and Utility. In *Uncertainty in Artificial Intelligence. Proceedings of the Eleventh Conference (1995)*, pages 3–10, San Francisco, 1995. Morgan Kaufmann Publishers.

[2] R. Bellman. *Dynamic Programming*. Princeton Univ. Press, Princeton, N.J., 1957.

[3] U. Bertele and F. Brioschi. *Nonserial Dynamic Programming*. Academic Press, 1972.

[4] D. Bertsekas. *Dynamic Programming and Optimal Control*, volume 2. Athena Scientific, 1995.

[5] D. Bertsekas and J. Tsitsiklis. *Neuro-Dynamic Programming*. Athena Scientific, 1996.

[6] D. Blackwell. Discounted Dynamic Programming. *Annals of Mathematical Statistics*, 36:226–235, 1965.

[7] V. Borkar. A convex analytic approach to Markov decision processes. *Probability Theory and Related Fields*, 78:583–602, 1988.

[8] Craig Boutilier. The `resource` problem. Personal communication, 2002.

[9] Craig Boutilier. The `robot` problem. Personal communication, 2002.

[10] Craig Boutilier, Thomas Dean, and Steve Hanks. Decision-theoretic planning: Structural assumptions and computational leverage. *Journal of Artificial Intelligence Research*, 11:1–94, 1999.

[11] Xavier Boyen and Daphne Koller. Tractable inference for complex stochastic processes. In *Proceedings of the Forteenth Annual Conference on Uncertainty in Artificial Intelligence*, pages 33–42, 1998.

[12] V. Chvatal. *Linear Programming*. W. H. Freeman and Company, 1983.

[13] G. B. Dantzig. *Linear Programming and Extensions*. University Press, 1963.

[14] G.B. Dantzig and P. Wolfe. Decomposition Principle for Linear Programs. *Operations Research*, 8:101–111, 1960.

[15] D. de Farias and B. Van Roy. The linear programming approach to approximate dynamic programming. *Operations Research*, 51(6):850–865, 2003.

[16] G. de Ghelink. Les Problèmes de décisions séquentielles. *Cahiers du Centre d'Etudes de Recherche Opérationelle*, 2:161–179, 1960.

[17] Thomas Dean and Keiji Kanazawa. Probabilistic temporal reasoning. In *Proceedings of the Seventh National Conference on Artificial Intelligence (AAAI-88)*, pages 524–528, St. Paul, Minnesota, 1988. American Association for Artificial Intelligence.

[18] Rina Dechter. Bucket elimination: A unifying framework for reasoning. *Artificial Intelligence*, 113(1-2):41–85, 1999.

[19] E.V. Denardo. On linear programming in a Markov decision problem. *Management Science*, 16(5):282–288, 1970.

[20] F. D'Epenoux. A probabilistic production and inventory problem. *Management Science*, 10(1):98–108, 1963.

[21] C. Derman. *Finite State Markovian Decision Processes*. Academic Press, New York, 1970.

[22] Judy Goldsmith, Michael L. Littman, and Martin Mundhenk. The complexity of plan existence and evaluation in probabilistic domains. In Dan Geiger and Prakash Pundalik Shenoy, editors, *Proceedings of the Thirteenth Annual Conference on Uncertainty in Artificial Intelligence*, pages 182–189, San Francisco, CA, 1997. Morgan Kaufmann Publishers.

[23] Judy Goldsmith and Martin Mundhenk. Complexity issues in markov decision processes. In *IEEE Conference on Computational Complexity*, pages 272–280, 1998.

[24] Judy Goldsmith and Pascal Poupart. The `advisor` problem. Unpublished manuscript, 2003.

[25] G. Gordon. *Approximate Solutions to Markov Decision Processes*. PhD thesis, Computer Science Department, Carnegie-Mellon University, Pittsburgh, Pennsylvania, 1999.

[26] Carlos Guestrin. *Planning Under Uncertainty in Complex Structured Environments*. PhD thesis, Computer Science, Stanford University, Stanford, California, 2003.

[27] Carlos Guestrin, Daphne Koller, and Ronald Parr. Max-norm projection for factored MDPs. In *Proceedings of the International Joint Conference on Artificial Intelligence*, 2001.

[28] Carlos Guestrin, Daphne Koller, and Ronald Parr. Multiagent planning with factored MDPs. In *Advances in Neural Processing Information Systems*, volume 14. The MIT Press, 2001.

[29] A. Hordijk and L.C.M. Kallenberg. Linear programming and Markov decision chains. *Management Science*, 25:352–362, 1979.

[30] R.A. Howard. *Dynamic Programming and Markov Processes*. MIT Press, 1960.

[31] N. Karmakar. A new polynomial-time algorithm for linear programming. *Combinatorica*, 4(4):373–395, 1984.

[32] L.G. Khachian. A polynomial algorithm for linear programming. *Soviet Math. Dokl.*, 20:191–194, 1979.

[33] V. Klee and G.J. Minty. How Good is the Simplex Algorithm? In O. Shisha, editor, *Inequalities, III*, pages 159–175. Academic Press, New York, NY, 1972.

[34] D. Koller and R. Parr. Computing factored value functions for policies in structured MDPs. In *Proceedings of the International Joint Conference on Artificial Intelligence*, 1999.

[35] D. Koller and R. Parr. Policy iteration for factored MDPs. In *Proceedings of the Sixteenth Annual Conference on Uncertainty in Artifical Intelligence*, 2000.

[36] H.J. Kushner and C.H. Chen. Decomposition of systems governed by Markov Chains. *IEEE Transactions on Automatic Control*, 5(19):501–507, 1974.

[37] Pierre Laroche, Yann Boniface, and René Schott. A new decomposition technique for solving markov decision processes. In *Proceedings of the 2001 ACM symposium on Applied computing*, pages 12–16. ACM Press, 2001.

[38] Michael L. Littman. Probabilistic propositional planning: Representations and complexity. In *Proceedings of the 14th National Conference on Artificial Intelligence*, pages 748–761, Providence, Rhode Island, 1997. AAAI Press / MIT Press.

[39] Michael L. Littman, Thomas L. Dean, and Leslie Pack Kaelbling. On the complexity of solving Markov decision problems. In *Proceedings of the Eleventh Annual Conference on Uncertainty in Artificial Intelligence*, pages 394–402, Montreal, Québec, Canada, 1995.

[40] Michael L. Littman, Judy Goldsmith, and Martin Mundhenk. The computational complexity of probabilistic planning. *Journal of Artificial Intelligence Research*, 9:1–36, 1998.

[41] C. Lusena, J. Goldsmith, and M. Mundhenk. Nonapproximability results for partially observable markov decision processes. *Journal of Artificial Intelligence Research*, 14:83–103, 2001.

[42] A.S. Manne. Linear programming and sequential decisions. *Management Science*, 6(3):259–267, 1960.

[43] M. Mundhenk, J. Goldsmith, C. Lusena, and E. Allender. Complexity of finite-horizon Markov decision processes. *Journal of the Association for Computing Machinery*, 47(4):681–720, 2000.

[44] C. Papadimitriou. *Computational Complexity*. Addison-Wesley, 1994.

[45] C.H. Papadimitriou and K. Steiglitz. *Combinatorial Optimization: Algorithms and Complexity*. Prentice-Hall, 1982.

[46] Christos H. Papadimitriou and John N. Tsitsiklis. The Complexity of Markov Decision Processes. *Mathematics of Operations Research*, 12(3):441–450, 1987.

[47] Relu Patrascu, Pascal Poupart, Dale Schuurmans, Craig Boutilier, and Carlos Guestrin. Greedy linear value-approximation for factored Markov decision processes. In *Proceedings of the Eighteenth National Conference on Artificial Intelligence (AAAI-02)*, pages 285–291, Edmonton, Alberta, Canada, 2002. AAAI Press.

[48] J. Pearl. *Probabilistic Reasoning in Intelligent Systems*. Morgan Kaufmann, 1988.

[49] J. D. Pearson. Duality and a decomposition technique. *SIAM-J-CONTROL*, 4(??):164–172, 1966.

[50] Pascal Poupart, Craig Boutilier, Relu Patrascu, and Dale Schuurmans. Piecewise Linear Value Function Approximation for Factored MDPs. In *Proceedings of the Eighteenth National Conference on Artificial Intelligence (AAAI-02)*, Edmonton, Alberta, Canada, 2002. AAAI Press.

[51] M. Puterman. *Markov Decision Processes: Discrete Dynamic Programming*. Wiley, 1994.

[52] Sheldon M. Ross. *Introduction to Probability Models*. Academic Press, sixth edition, 1997.

[53] Dale Schuurmans and Relu Patrascu. Direct value-approximation for factored MDPs. In *Advances in Neural Processing Information Systems*, volume 14. The MIT Press, 2001.

[54] P. Schweitzer and Abraham Seidmann. Generalized polynomial approximations in Markovian decision processes. *J. Math. Analysis and Appl.*, 110:568–582, 1985.

[55] Satinder P. Singh, Tommi Jaakkola, and Michael I. Jordan. Reinforcement learning with soft state aggregation. In G. Tesauro, D. Touretzky, and T. Leen, editors, *Advances in Neural Information Processing Systems*, volume 7, pages 361–368. The MIT Press, 1995.

[56] R. St-Aubin, J. Hoey, and C. Boutilier. APRICODD: Approximating policy construction using decision diagrams. In *Advances in Neural Information Processing Systems*, volume 13. The MIT Press, 2000.

[57] M. A. Trick and S. E. Zin. A linear programming approach to solving stochastic dynamic programs. Unpublished manuscript, 1993.

[58] M. A. Trick and S. E. Zin. Spline approximations to value functions: A linear programming approach. *Macroeconomic Dynamics*, 1:255–277, 1997.

[59] R. Williams and L. Baird. Tight performance bounds on greedy policies based on imperfect value functions. Technical report, Northeastern University, 1993.